GDBOD: Density-Based Outlier Detection Exploiting Efficient Tree Traversals on the GPU

Revanth Reddy Munugala
School of Informatics, Computing, and Cyber Systems
Northern Arizona University
Flagstaff, AZ, U.S.A.
rm2878@nau.edu

Michael Gowanlock
School of Informatics, Computing, and Cyber Systems
Northern Arizona University
Flagstaff, AZ, U.S.A.
michael.gowanlock@nau.edu

***Abstract*—Outlier detection algorithms are employed across numerous application domains. In contrast to distance-based outlier detection algorithms that compute distances between points, hypercube-based algorithms reduce computational costs by evaluating the density of a point based on its enclosing hypercube. A major limitation of state-of-the-art hypercube-based algorithms is that they do not scale to large datasets. This paper proposes GPU Density-Based Outlier Detection (GDBOD) that is supported by efficient tree-based hypercube search methods. We propose two GPU-friendly n-ary tree data structures for efficient hypercube searches which are optimized to obtain good locality and exploit the fine-grained parallelism afforded by the GPU. Also, we propose a data encoding method that compresses data to reduce the number of comparisons during distinct hypercube array construction and reorder the coordinates of the input dataset to enhance neighborhood search performance. Additionally, we design sequential and multi-core CPU algorithms that can be employed on systems not equipped with GPUs. Our sequential CPU algorithm achieves a mean speedup of 18.35× over the state-of-the-art and our parallel GPU algorithm achieves a mean speedup of 3.29× over our multi-core CPU algorithm across 6 real-world datasets. With our proposed optimizations on the GPU, we achieve a peak compute throughput of 86.51%, along with 92.06% L1 cache hits and 92.94% L2 cache hits.**

***Index Terms*—Data Analytics, GPU, In-memory Databases, Outlier Detection**

I. Introduction

Outlier detection (OD) finds objects that deviate from the majority of the other objects in a dataset. OD is used in many contexts, such as health care, fraud detection, cyber security, surveillance, sensor networks, data quality and cleaning, time-series monitoring, and the Internet of Things [1], [2]. Given the large number of domains that rely on OD, fast and scalable OD methods are of great importance.

There are two major classes of OD that are described as follows: (i) distance-based OD [3]; and, (ii) hypercube-based OD [4], [5]. *Distance-based approaches:* an outlier is defined as a point that has a substantial fraction of a dataset exterior to a search distance around the point [6], or is defined using properties of its nearest neighbors [7]. *Hypercube-based approaches:* an outlier is defined based on each point's local density in the data space. To differentiate between the distance-based approaches above, here we refer to approaches that compute the density *without* relying on distance calculations. Therefore, these approaches do not perform neighborhood searches/range queries. This approach is more computationally efficient than distance-based approaches.

Cabral & Cordeiro [8] proposed HySortOD which assigns data points to hypercubes, which are axis-aligned partitions in d dimensions. These hypercubes are used to assign each point a density based on its parent hypercube. In the supervised context, HySortOD outperforms state-of-the-art algorithms in terms of accuracy; however, the algorithm is limited to processing small datasets.

To address the small dataset limitation described above, we propose a GPU-accelerated algorithm, GPU Density-Based Outlier Detection (GDBOD), that addresses large datasets. GPUs have architectural features that make them well-suited for data analytics applications, including OD. GPUs are equipped with high memory bandwidth and have thousands of cores which are best exploited for data-parallel algorithms. Given that the algorithm spends significant time finding adjacent hypercubes — which is clearly a data-parallel task — it is well-suited to GPU acceleration. To this end, this paper makes the following contributions.

- We design two GPU-friendly tree-based data structures that exploit locality that support hypercube searches. The algorithm employs compression on hypercube data and reorders the coordinates of each data point to increase the probability of tree traversals short-circuiting during a depth-first search.
- We exploit the GPU's fine-grained parallelism by having multiple threads find adjacent hypercubes of a query hypercube. This decreases load imbalance within warps[1].
- We also design sequential and parallel multi-core CPU algorithms. We show that the sequential algorithm has superior performance to the state-of-the-art algorithm [8].
- Our evaluation employs six real-world datasets spanning $d \in [18, 128]$ dimensions with 500K–11.6M data points.

Paper organization: Section II outlines the background and related work. Section III outlines our algorithm (GDBOD). Section IV presents the experimental evaluation, and Section V concludes the paper.

This material is based upon work supported by the National Science Foundation under Grant No. 2042155.

[1]We use CUDA terminology throughout this paper.

II. Background

A. Related Work

Distance-based GPU Approaches: OD algorithms aim to identify anomalies in a given data space. Distance-based algorithms identify outliers by quantifying the distances between data points. These algorithms identify outliers by assuming that typical points in the data are close to each other, whereas outliers are far from the majority. Nevertheless, computing distances can be resource-intensive, particularly when processing large high-dimensional datasets. Example contributions to GPU distance-based OD include: Angiulli et al. [9] which proposed a family of parallel distance-based outlier detection algorithms for the GPU, Matsumoto and Hung [10] proposed a fast parallel outlier detection in uncertain data, and HewaNadungodage et al. [11] proposed a GPU algorithm for outlier detection for continuous data streams.

Hypercube-based CPU Approaches: To mitigate the computational costs associated with computing distance calculations, CPU OD algorithms such as HySortOD [8], DB-Out [12], and aLOCI [13] adopt a hypercube-based approach. Despite eliminating distance calculations, hypercube-based OD still requires efficient hypercube searches. Neighborhood searches are data-parallel tasks that are well-suited to GPU architecture.

GPU Tree-based Searches: Canonical data structures for finding nearby hypercubes to a query hypercube typically use trees. However, the limitations of GPU tree searches include limited stack space, branch divergence, and load imbalance [14]. Also, traversals of tree data structures rely on GPU task parallelism to process a batch of queries, which may cause warp divergence as each thread processes a different traversal [15]. To address the limited stack space issue, Goldfarb et al. [16] proposed a stackless tree algorithm using auxiliary pointers that can be applied to all recursive tree traversals on the GPU. To address warp divergence on the GPU, the Parallel Scanning and Backtracking (PSB) algorithm was proposed by Nam et al. [14]. Zhang et al. [17] proposed the RegTT tree traversal algorithm that reorders queries based on the truncation history to mitigate load imbalance for tree traversals on the GPU. Merrill et al. [18] proposed a parallelization strategy for BFS traversals on the GPU. Our approach addresses all of these facets of efficient GPU tree traversals — it employs (i) a stackless traversal algorithm, (ii) that addresses warp divergence that yields negligible load imbalance; and, (iii) has excellent locality.

B. State-of-the-art: HySortOD

HySortOD proposed by Cabral & Cordeiro [8] is a hypercube-based OD algorithm for the CPU where data points are mapped to bounded multi-dimensional regions denoted as hypercubes. The hypercubes are then sorted based on their coordinates such that they are positioned near each other in memory. A tree-based approach is used to calculate the neighborhood density of the hypercubes and then the outlierness score of each point is calculated where all points within the same hypercube have the same score. This general outline applies to GDBOD presented in Section III.

1) Hypercube Construction & Sorting: In a normalized dataset D having d dimensions, the coordinates of each point lie in the range $[0, 1]$. We define a point $p_i \in D$ as $p_i = \{x_i^1, x_i^2, \ldots, x_i^d\}$. Each point p_i in the dataset is mapped to a d dimensional hypercube $h_i = \{\lfloor x_i^1/l \rfloor, \lfloor x_i^2/l \rfloor, \ldots, \lfloor x_i^d/l \rfloor\}$ where l is the length of partition in each dimension. The length of the partition is determined by the number of bins $b \in \mathbb{N}_{>1}$ where $l = \frac{1}{b}$. The constructed hypercubes are stored in an array $H = [h_1, h_2, ..., h_{|H|}]$ and the number of points in each hypercube is stored in an array $C = [c_1, c_2, ..., c_{|H|}]$ where C_m is the number of points of a hypercube h_m for $m = 1, 2, \ldots, |H|$. The distinct hypercubes in H are sorted using lexicographical ordering as a function of increasing dimensionality to bring the neighboring hypercubes close to each other in the search space.

2) Neighborhood Search: We define hypercubes using the notation $h_i = \{h_i^1, h_i^2, \ldots, h_i^d\}$, where $h_i \in H$. A hypercube h_i is defined as an immediate neighbor of hypercube h_j in the hypercube array H iff $|h_i^q - h_j^q| \leq 1$ where $1 \leq q \leq d$. Here, h_i^q and h_j^q are the coordinate values of dimension q of hypercube h_i and h_j, respectively.

The immediate neighbors of a given hypercube h_i can be identified by employing a linear scan to search all the hypercubes in the hypercube array H. To improve performance, a tree-based structure is leveraged to effectively find the immediate neighbors of h_i by limiting comparisons between h_i and all other hypercubes $h_j \in H$.

Every node in the first level of the tree structure stores the starting and ending indices of a subset of hypercubes in H that have the same coordinate value in the first dimension. For each node in the previous level, the next level of the tree structure is constructed by repeating the same process until all the dimensions are mapped. A threshold parameter *MinSplit* is defined to limit the number of mapped hypercubes, as the linear scan of hypercubes might be faster than the tree traversal depending on the data distribution. *MinSplit* limits tree depth and node traversal for immediate neighbor identification without impacting the outlierness score. The tree nodes are further mapped in subsequent dimensions only if the number of hypercubes mapped in the current dimension is at least equal to *MinSplit*. This parameter reaches a trade-off between a tree traversal and a scan over H.

3) Neighborhood Density & Outlierness Score: The neighborhood density (w_i) of a hypercube h_i is defined as the sum of the number of points of all its immediate neighbors and is defined as follows: $w_i = \sum_{h_m \in N(h_i)} C_m$. Here, w_i is the neighborhood density of the hypercube h_i, and $N(h_i)$ is the set of all immediate neighbors of hypercube h_i. The neighborhood densities of all hypercubes in the hypercube array H are stored in an array $W = [w_1, w_2, ..., w_{|H|}]$. The outlierness score of a hypercube h_i is calculated by normalizing it to the maximum neighborhood density w_{max} in W as follows: $O_i(w_i, w_{max}) = 1 - (w_i/w_{max})$. Scores approaching zero indicate an inlier and scores close to one indicate an outlier.

III. GDBOD: GPU Density-Based Outlier Detection

In this section, we describe our proposed GPU algorithms which include a naive scan-based algorithm (GPU-Naive), a dimensional-level tree-based algorithm (GPU-DL-Tree), and a locality and traversal revised tree-based algorithm (GPU-LTR-Tree). As part of each algorithm, we outline various optimizations for different algorithm components including hypercube construction and tree traversals. We assume that the hypercube data can fit within the global memory. We require less memory because we use hypercubes as references instead of actual data points to identify outliers. Due to space constraints, we do not describe our parallel CPU algorithm that is also evaluated in Section IV. However, the algorithm descriptions in the subsequent sections can be applied to both CPU and GPU algorithms, but the implementation details including GPU optimizations differ significantly.

A. Naive Scan-based Algorithm

The immediate neighbors of a given hypercube $h_i \in H$ can be found by scanning through all the hypercubes in hypercube array H and validating if it meets the criteria for immediate neighbors described in Section II-B2. In subsequent sections, we outline the construction of a distinct sorted hypercube array and naive scan-based algorithm (GPU-Naive) to identify the immediate neighbors of a given hypercube along with its corresponding GPU optimizations.

1) Hypercube Construction: The dataset D of $|D|$ points is stored in row-major order where $D = [p_1, p_2, \ldots, p_{|D|}]$. Hypercube array H is constructed in parallel by computing the assignment of each point $p_i \in D$ to its hypercube as described in Section II-B1. We only store non-empty hypercubes in H to limit memory consumption. In the following sections, we describe an optimization technique, referred to as EncDec, that efficiently constructs the distinct hypercube array H.

2) Hypercube Encoding: A hypercube of d dimensions requires d elements to represent its coordinates. The total number of elements needed to store indistinct hypercube data of a dataset D of $|D|$ points is $|D| \cdot d$. The number of comparisons required to check if a hypercube h_i is already present in the distinct hypercube array H is $|H| \cdot d$. This operation is expensive for large datasets, so we use an optimization that efficiently stores the hypercube data and reduces the comparisons while removing duplicates using bit shifting.

Algorithm 1 outlines our encoding technique (EncDec) on the GPU where the coordinates of a hypercube are encoded using bit shifting to compress the data into fewer data chunks. Without the loss of generality, we illustrate encoding and decoding hypercube coordinates using a 64-bit unsigned integer. Assuming that a coordinate h_i^j requires at least k bits, the length of the array that needs to be allocated to store the coordinates of d dimensions of a hypercube h_i is $l_{enc} = \lceil 64/k \rceil$. This optimizes memory usage and improves performance while removing duplicate hypercubes.[2]

Algorithm 1 Encoding Algorithm.

```
1:  procedure ENCODE(k, d, h_i, h')
2:      maxCount ← ⌈64/k⌉                          ▷ Max count per data type.
3:      encodeCount ← 0, idx ← 0
4:      for j ∈ 1, 2, ..., d do
5:          if encodeCount = maxCount then
6:              idx ← idx + 1, encodeCount ← 0      ▷ Update counters.
7:          end if
8:          h'[idx] ← h'[idx] ≪ k ∨ h_i^j           ▷ Encode current dimension.
9:      end for
10:     return h'
11: end procedure
```

We describe Algorithm 1 as follows: line 2 sets the value for the maximum number of dimensions per 64-bit unsigned integer. Within the loop body on line 4, the index of the current block of the 64-bit unsigned integer is updated to the next position if the current block is filled. On line 8, the value of h_i^j is encoded into $h'[idx]$ by left shifting the existing value in $h'[idx]$ by k bits and performing a bitwise OR operation with h_i^j where $h'[idx]$ is the idx^{th} value in encoded hypercube h'.

3) Removing Duplicates and Sorting Hypercubes: The encoded hypercube array H' is copied from the GPU's device memory to the host. On the CPU, an ordered map M is used to store the encoded hypercube as a key and the array of dataset points belonging to the hypercube as its corresponding value. If a hypercube is not present in the map M, a new key-value pair is appended to the map, otherwise, the array of points in the hypercube is updated. Usage of an ordered map removes the duplicates in the hypercube array and the hypercube array is sorted using the lexicographical order as a function of increasing dimensionality which brings the neighboring hypercubes close to each other in memory. Encoding the hypercube data reduces the number of comparisons required to check if a hypercube is already present in the map M and is an effective optimization for datasets with high dimensionality.

4) Decoding the Encoded Hypercube Array: The distinct encoded hypercube data is copied to a 1-D array H' and is transferred to the device for decoding. On the GPU, the encoded hypercube array H' is decoded in parallel to construct a distinct hypercube array H.

Our decoding algorithm closely resembles our encoding technique. Dimension d of an encoded hypercube $h' \in H'$ is extracted by performing a bitwise AND operation with k bits set to 1 on the last block of the encoded hypercube. Subsequently, the encoded data is right-shifted by k bits to extract the next dimension, $d - 1$. This sequential process continues until all dimensions are extracted. The decoded hypercube h_i is then stored in an array of distinct hypercubes $H = [h_1, h_2, ..., h_{|H|}]$. Figure 1 illustrates the various steps involved in constructing a distinct sorted hypercube array H and the architecture in which the construction takes place.

[2]The coordinates of the dimensions of a hypercube with $b = 12$ lie within $[0, 12]$. The binary representation of the maximum dimension 12 is 1100 and 4 bits are required to store the value of one hypercube coordinate. An unsigned 64-bit integer can store $\lceil 64/4 \rceil = 16$ dimensions.

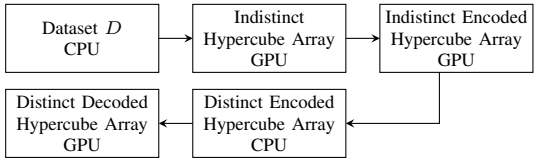


Fig. 1. Construction of distinct hypercube array H.

Algorithm 2 Tree Construction of GPU-DL-TREE.

```
1:  procedure BUILDTREE(H, T, m, curDim)
2:      if curDim > d then return
3:      end if
4:      dimStart ← getDimStart(), dimEnd ← getDimEnd()
5:      for i ∈ [dimStart, dimEnd] do
6:          start ← T_i^start, end ← T_i^end                ▷ Get start and end.
7:          if end − start + 1 ≥ m then
8:              val ← h_i^{curDim∓1}, childCount ← 0
9:              for j ∈ [start, end] do
10:                 if h_j^{curDim+1} ≠ val then
11:                     appendNode(T_n, T)          ▷ Append new tree node T_n to T.
12:                     T_n^start ← start, T_n^end ← j − 1, T_n^coordinate ← val
13:                     updMap()          ▷ Update parent, sibling, and child mapping.
14:                     start ← j, val ← h_j^{curDim+1}          ▷ Update start and val.
15:                 end if
16:             end for
17:         end if
18:     end for
19:     buildTree(H, T, m, curDim + 1)
20:     return T
21: end procedure
```

5) Neighborhood Search: In GPU-NAIVE, the neighborhood density of a hypercube $h_i \in H$ is calculated by performing a linear scan over all the hypercubes in the distinct hypercube array H constructed in Section III-A4, validating if the target hypercube satisfies the definition of an immediate neighbor, and updating the neighborhood density w_i.

B. GPU-DL-TREE

The drawback of the naive approach is that the linear scan is expensive, especially for large datasets. The time complexity to identify immediate neighbors for all hypercubes in H is quadratic, $O(|H|^2)$. The number of comparisons to identify immediate neighbors of a hypercube $h_i \in H$ can be reduced by employing a tree-based structure. Throughout the paper, we reuse the ENCDEC optimization technique for hypercube array construction described in Section III-A, and we propose a tree-based approach for the GPU (GPU-DL-TREE) to improve the performance of the neighborhood search.

1) Tree Construction: In this section, we describe the construction of a n-ary dimension-wise tree structure (GPU-DL-TREE), denoted as T, to improve the performance of the neighborhood search. Each tree node $T_i \in T$ is defined by five attributes: coordinate value ($T_i^{coordinate}$), starting index (T_i^{start}), and the ending index of the shared coordinate (T_i^{end}), sibling index ($T_i^{sibling}$), child index (T_i^{child}), and parent index (T_i^{parent}). Here, the parent index in a node points to a tree node of the prior dimension level from which the current node is mapped. The child index points to the index of the first child node mapped from the current node and the sibling index points to the next node mapped from the same parent.

Algorithm 2 details the construction of GPU-DL-TREE sequentially on the CPU by mapping hypercubes sharing the same coordinate in the first dimension and storing them in an array T. The hypercubes of subsequent dimensions are recursively mapped based on the starting and ending index of the coordinate value of each node in the previous dimension level and are stored contiguously.

Initially, the root node that does not map any coordinate is appended to the tree array T with the coordinate, starting, and ending index of the root node as ϕ, 1, and $|H|$, respectively. Line 4 in Algorithm 2 store the starting index and the ending index of the tree nodes belonging to the current dimension in the tree array T. For each tree node in the current dimension, line 7 ensures that the number of hypercubes mapped by the current tree node is at least equal to the threshold parameter *MinSplit* (m) for further mapping. The coordinate value of the next dimension of the hypercubes is scanned within the starting and ending index of the tree node to identify a change in the coordinate value. Whenever a change takes place, a new tree node T_n with the current coordinate value, starting index, and ending index within which the coordinate value remains the same is added to the array. The parent and child mapping is updated between the appended node T_n and the tree node T_{parent} from which the current node is mapped. Similarly, sibling mapping is updated when multiple nodes are mapped from the same parent node (line 13). The tree is recursively constructed by appending the nodes of the current level and reiterating to append the nodes of the next level (line 19).

Illustrative example of the tree: In Figure 2(a), we illustrate a tree constructed based on the 2-dimensional sorted hypercube array $H = [(0,0),(0,1),(1,1),(2,0),(2,1)]$ using *MinSplit* parameter $m = 2$. The number of hypercubes mapped by the tree node T_2 is 1, which is below the threshold $m = 2$ and hence it does not undergo further mapping. Here, for tree node T_1, the parent index is T_0, the sibling index is T_2 and the child index is T_4. The tree is stored in a 1-D array, T, as shown in Figure 2(b), where the tree nodes that map the coordinates of the same dimension are stored together.

2) Tree Traversal: This section outlines the GPU-DL-TREE traversal constructed in Section III-B1. Tree traversals occur iteratively on the GPU instead of using recursion due to limited stack space. The tree traversal is split into three parts: movement to (i) the child node, (ii) the sibling node, and (iii) the parent node. We describe the traversal as follows.

1) If the current tree node has a child node, then the current index is updated to the index of the child node.
2) If the current tree node does not have a child node and has a sibling node, then the current index is updated to the index of the sibling node.
3) If the current tree node does not have a child and sibling node, then the current index is updated to the parent node until the current node has a sibling node. Then, the current index is updated to the sibling index of the current node.
4) Steps 1–3 are repeated until the entire tree is traversed.

Figure 2(c) represents the tree traversal starting from root node T_0. Root node T_0 has a child node T_1 and the current

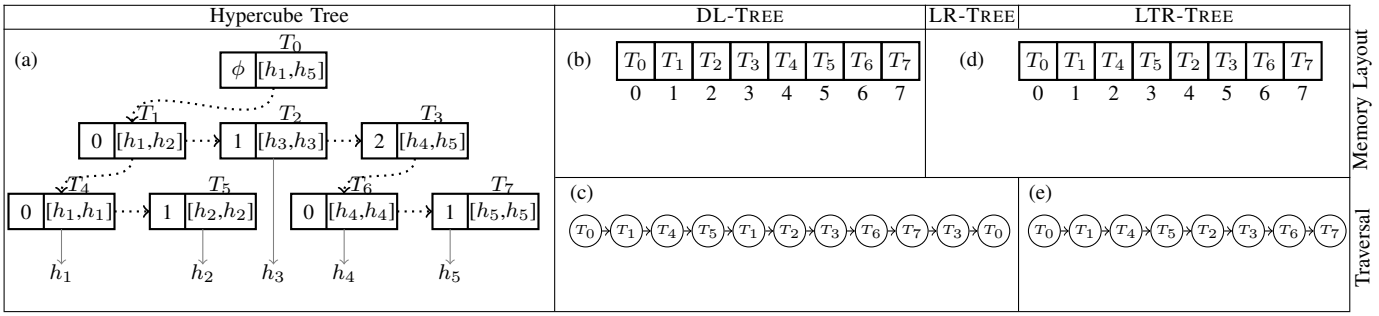


Fig. 2. (a) Hypercube tree constructed using dataset D with $d = 2$ and $m = 2$. Tree nodes are represented as T_i, dotted lines indicate the tree traversal while calculating neighborhood density. (b) Tree node positions of GPU-DL-TREE based on (a). (c) Tree traversal on the GPU-DL-TREE based on node positions in (b). (d) Revised node positions of GPU-LR-TREE based on (a). (e) Tree traversal on GPU-LTR-TREE based on node positions in (d).

Algorithm 3 Neighborhood Search of GPU-DL-TREE.

```
1:  procedure NEIGHBORHOODSEARCH(T, H, h_i, W, C, startIdx, curDim)
2:      cur ← startIdx, end ← T_cur^sibling          ▷ Set starting and ending indices.
3:      while cur ≠ end ∧ cur ≠ φ do
4:          if T_cur is leaf node then
5:              for j ∈ [T_cur^start, T_cur^end] do
6:                  if h_i is immediate neighbor of h_j then
7:                      W_i ← W_i + C_j              ▷ Update neighborhood density.
8:                  end if
9:              end for
10:         end if
11:         if T_cur^child ≠ φ ∧ T_cur is a neighbor of h_i then
12:             tmp ← T_cur^child
13:             while T_tmp is not a neighbor of h_i do
14:                 tmp ← T_tmp^sibling
15:             end while
16:             if tmp ≠ φ then
17:                 cur ← tmp, curDim ← curDim + 1
18:             else
19:                 cur ← getNextIndex(T, cur, curDim)      ▷ Backtrack.
20:             end if
21:         else if T_cur^sibling ≠ φ then          ▷ Check if current node has a sibling.
22:             cur ← T_cur^sibling
23:         else
24:             cur ← getNextIndex(T, cur, curDim)      ▷ Backtrack.
25:         end if
26:     end while
27:     return W
28: end procedure
```

node pointer moves to the node T_1 and moves to node T_4 as T_4 is the child of current node T_1. The node T_4 does not have a child node but has a sibling node T_5, so the current node pointer moves to node T_5. Node T_5 does not have a child node and a sibling node, so it moves back to the parent node T_1 and to its sibling node T_2. The same steps described above are repeated until the the entire tree is traversed.

3) Neighborhood Search: The tree T is transferred from the host to the device for the calculation of the neighborhood density. Algorithm 3 outlines the neighborhood search used to calculate the neighborhood density of a hypercube h_i on the GPU. To describe the neighborhood search, we define a tree node T_k as a neighbor of h_l at dimension j if $|h_l^j - h_k^j| \leq 1$ where $h_k \in T_k$. While traversing the tree to calculate the neighborhood density of a hypercube h_i, the start index is set as the index of the root node, and the end index is set as the sibling index of the starting node (line 2). Also, we investigate assigning multiple threads to calculate the neighborhood density of h_i (Section III-F).

While traversing the tree for calculating the neighborhood density of a hypercube h_i, if the current node is a leaf node, the hypercubes mapped within the leaf node are validated if they are immediate neighbors. If they are, the neighborhood density of the hypercube h_i is updated by adding the number of points in each hypercube (Algorithm 3, lines 4–7).

In Algorithm 3, on line 11, if the current node T_{cur} has a child node T_{child} and T_{cur} is a neighbor of h_i, the current node index is updated with the index of T_{child} if T_{child} is also a neighbor of h_k to ensure that the child node is also an immediate neighbor in the current dimension. If T_{child} is not a neighbor of h_i, then the sibling nodes of T_{child} are traversed to find a node $T_{sibling}$ that is a neighbor of h_i. Then the current node index is updated with the index of $T_{sibling}$ (lines 13–14). If none of the tree nodes mapped by the node T_{cur} is a neighbor of h_i, then the current index is updated with the index of the parent node of T_{cur} until the parent node T_{parent} has a sibling node. The current index is then updated with the index of the sibling node of T_{parent} (line 19). If T_{cur} does not have a child node or is not a neighbor of h_i but has a sibling node $T_{sibling}$ (line 21), the steps detailed in Section III-B2 are followed to traverse the entire tree T.

C. GPU-LR-TREE

Motivation: One of the limitations of the tree traversal described in Algorithm 3 is that the memory accesses to the nodes in the array T may not have good locality due to branching. The branching on the GPU architecture results in a loss of parallel efficiency due to the serialization of instructions in a warp. Another issue with the traversal is that the same node is accessed multiple times while backtracking to the parent node. In the subsequent sections, we propose another tree that addresses possible inefficiencies during tree traversals.

Recall that the nodes in GPU-DL-TREE are arranged in the tree T as shown in Figure 2(b). To address the drawbacks of the GPU-DL-TREE, we detail GPU-LR-TREE where nodes are rearranged without altering the node mapping.

The memory accesses to the nodes during the depth first search tree traversal described in Algorithm 3 can be optimized by rearranging the nodes in the array such that the child node is close to its parent node in memory. This can be achieved by creating a new tree with the same number of elements

as T and placing the nodes in the new tree T' in the same order as they are accessed in the GPU-DL-TREE using the same tree traversal described in Section III-B2. The child, parent, and sibling mapping of the nodes placed in the new positions is revised by keeping track of the current node T'_{cur} and the last visited node T'_{old}. The following steps describe the construction of the optimized tree (T').

1) If T'_{cur} is the child of T'_{old}, update the parent and child indices of T'_{cur} and T'_{old} with the node positions of T'_{cur} and T'_{old} respectively.
2) If T'_{cur} is the sibling node of T'_{old}, update the sibling index of T'_{old} with the index of T'_{cur}.
3) While backtracking to the parent node of T'_{cur} when it does not have child and sibling nodes, update the parent mapping of T'_{cur} with the index of the parent of T'_{cur}.
4) Repeat steps 1–3 until the whole tree is traversed.

Figure 2(d) shows the node positions of GPU-LR-TREE. The tree traversal for the neighborhood search procedure in GPU-LR-TREE is identical to that of GPU-DL-TREE.

D. GPU-LTR-TREE

Another potential issue with the tree traversal defined in Algorithm 3 is that each node might be accessed multiple times while backtracking to the parent node which may directly impact performance, particularly on high dimensional datasets where the cost of backtracking is high. To ensure that each node in the tree is accessed only once, the earlier node structure is revised which removes the need for the parent and sibling indices in the tree node structure. We denote this new index as the "break index". The break index points directly to the next node during the tree traversal if the current node does not have a child node. For example, in Figure 2(a), node T_5 does not have a child or sibling node, and hence, the break index value of T_5 is the index of T_2 which is the first sibling node of the parent node of T_5. In subsequent sections, we describe the construction and neighborhood search for the locality and traversal revised tree (GPU-LTR-TREE).

1) Tree Construction: The GPU-LTR-TREE node positions are identical to that of GPU-LR-TREE as described in Section III-C. The construction of the GPU-LTR-TREE, denoted as T'', using the new node structure is achieved by copying the coordinate value, start index, end index, and child node values from the old node structure (GPU-LR-TREE) to the new node structure (GPU-LTR-TREE) following the previously defined tree traversal described in Section III-B2. The break index of the nodes of GPU-LTR-TREE T'' is updated as follows:

1) If the current node in T'' has a sibling node, then the break index of current node in T'' is the index of the $T''_{sibling}$.
2) If the current node T''_{cur} does not have a sibling node, then the break index of the current node in T'' is the break index of the parent node of the current node in T''.

To construct GPU-LTR-TREE, the following tree constructions are required in this order: GPU-DL-TREE (T) $\rightarrow$ GPU-LR-TREE (T') $\rightarrow$ GPU-LTR-TREE (T''). Every node in GPU-LTR-TREE contains a break index that points to the next available node when the current node does not have a child node or when the current node does not satisfy the definition of an immediate neighbor for a hypercube h_i. This removes the need for backtracking to the parent node to find other available nodes and ensures that each node is visited only once while traversing the tree during the neighborhood search.

2) Neighborhood Search: We outline the neighborhood search for GPU-LTR-TREE (T'') as follows.

1) If the current node is a leaf node, then the hypercubes within the node are validated to check if they are the immediate neighbors of hypercube h_i and the neighborhood density w_i is updated.
2) If the current node has a child node and is the immediate neighbor of a target hypercube h_i, then the current index is updated with the index of the child node.
3) If the current node does not have a child node or is not an immediate neighbor of h_i, then the current index is updated with the index of the break index of the current node.
4) Steps 1–3 are repeated until the whole tree is traversed.

Figure 2(e) shows the traversal of the hypercubes in Figure 2(a) where each node is accessed once and has improved locality relative to the tree traversal shown in Figure 2(c).

E. Reordering of Dataset Point Coordinates

GPU-DL-TREE, GPU-LR-TREE, and GPU-LTR-TREE are constructed based on the coordinates in increasing order of dimensions $1, 2, \ldots, d$. However, the performance of the neighborhood search is sensitive to the order of the dimensions in the dataset and can be improved by reordering the coordinates of the points based on the variance of the coordinate values in each dimension [19], [20].[3]

Consider that dimensions with higher variance imply that there is significant spread in the data whereas low variance in a dimension implies that the data are clustered in that dimension. Thus, if the dimensions with the greatest variance are located in the first few layers of the tree, then it is more likely to short circuit early (aborting the search) during the tree traversal. We exploit this property to short circuit early, by reordering the coordinates of $p_i \in D$ from highest to lowest variance. We refer to reordering the dimensions as REORDER and NOREORDER when the optimization is disabled.

Using REORDER has better performance relative to NOREORDER due to improved short-circuiting during the tree traversal. However, performance gains are not guaranteed because the optimization depends on the data distribution of the dataset. To address this issue, we assess the dataset's dispersion by computing the coefficient of variance of the variance of each of the dimensions and only reorder the dimensions if and only if there is a high dispersion. The coefficient of variance is defined as the ratio of the standard deviation of the variance of the dimensions to the mean variance.

Based on the above, we propose the SELECTIVEREORDER optimization to selectively reorder the dimensions if the coef-

[3]For example, consider a $d = 3$ point $p_a = \{1, 2, 3\}$. We can swap dimensions 1 and 3 to obtain $p_a = \{3, 2, 1\}$. If we swap the dimensions 1 and 3 for all $p_i \in D$, then the result of the algorithm is the same.

TABLE I
SUMMARY OF THE ALGORITHMS AND THEIR OPTIMIZATIONS.

Optimization	GPU-DL-TREE	GPU-LR-TREE	GPU-LTR-TREE	GPU-NAIVE
Locality	✗(Sec. III-B1)	✓(Sec. III-C)	✓(Sec. III-D1)	N/A
Traversal	✗(Sec. III-B2)	✗(Sec. III-C)	✓(Sec. III-D2)	N/A
Fine-Grained Parallelism	✓(Sec. III-F)	✓(Sec. III-F)	✓(Sec. III-F)	✓(Sec. III-F)
SELECTIVEREORDER	✓(Sec. III-E)	✓(Sec. III-E)	✓(Sec. III-E)	✓(Sec. III-E)

ficient of variance reaches a threshold c. A higher coefficient of variance implies that the dispersion of the variance of dimensions in the dataset is high which directly correlates to greater performance gains due to the REORDER optimization. We will compare the performance of the NOREORDER and REORDER optimization techniques, and show c can be selected using empirical results.

F. Fine-Grained Parallelism for the GPU

There is an opportunity to increase fine-grained parallelism during neighborhood searches for the GPU approaches GPU-NAIVE and the GPU trees (GPU-DL-TREE, GPU-LR-TREE, and GPU-LTR-TREE). The baseline GPU-NAIVE and GPU trees above use a total of $|H|$ GPU threads to identify the immediate neighbors of $h_i \in H$. To saturate GPU resources (compute cores), it is critical that we create additional threads, particularly if the number of hypercubes $|H|$ is small. We employ additional threads for the algorithms as follows:

GPU-NAIVE: For a given hypercube search, we assign multiple threads (t) to perform a linear scan over a subset of H, where each thread computes $|H|/t$ elements.

GPU trees: Similarly, we allocate multiple threads (t) to each hypercube being searched (h_i) to traverse the tree. The number of threads, t, is based on the number of tree nodes in the first dimension of the tree. Typically, $t = b$ for real-world datasets.

G. Summary

Table I summarizes the proposed algorithms, their optimizations, and corresponding sections in the paper. We compare two data structures for hypercube searches: GPU-DL-TREE and GPU-LTR-TREE. Tree nodes in GPU-DL-TREE are stored such that those with the same dimension are close to each other in memory. In contrast, the nodes in GPU-LTR-TREE are reordered such that the child node is closer to the parent node. This allows comparing different traversal patterns and their impact on performance.

IV. EXPERIMENTAL EVALUATION

A. Experimental Methodology

Table II outlines the hardware platforms that we employ. All host and device code is compiled with g++ and CUDA v 12.2, respectively, and uses the O3 compiler optimization flag. To understand the effects of different GPUs on the performance of our algorithms we use the GPUs shown in PLATFORMA (A100) and PLATFORMB (Quadro RTX 5000) which are datacenter- and consumer-grade GPUs, respectively. Other details can be found in Table II. All parallel CPU experiments are executed on PLATFORMA (using OpenMP).

TABLE II
THE PLATFORMS USED IN OUR EXPERIMENTAL EVALUATION. PLATFORMA CONTAINS A DATACENTER-GRADE GPU AND PLATFORMB CONTAINS A CONSUMER-GRADE GPU.

Platform	Model	Total Cores	Clock (GHz)	Main/ Global Memory
CPU				
PLATFORMA	2×AMD Epyc 7542	64	2.9	512 GiB
PLATFORMB	Intel W-2295	18	3.0	256 GiB
GPU				
PLATFORMA	A100	6912	1.4	40 GiB
PLATFORMB	Quadro RTX 5000	3072	1.8	16 GiB

TABLE III
DATASETS USED IN THE EXPERIMENTAL EVALUATION ORDERED BY DIMENSIONALITY, d. THE DATASET SIZE IS $|D|$.

Dataset	d	$\|D\|$
SuSy	18	5,000,000
Hepmass	27	10,500,000
Higgs	28	11,000,000
BigCross	57	11,620,300
MSD	90	515,345
Sift	128	10,000,000

The time needed to load the dataset is excluded in all experiments. All other time components, including tree construction, data transfer between host and GPU, and optimizations are included in the response time. Due to significant response times on the CPU, we impose a time limit of 16 hours for all experiments. We benchmark the algorithms using unsupervised mode without computing AUC scores because the datasets in Table III do not have class labels. The time to compute the AUC score is negligible and thus does not affect our results. The source code for GDBOD is publicly available.[4]

B. Datasets

We use the real-world datasets listed in Table III, containing 500K–11.6M data points with 18–128 dimensions.[5] We do not use the labeled datasets used in the accuracy evaluation of the HySortOD [8] as they are too small to evaluate the performance of our GPU algorithms. However, we compare the performance of the HySortOD [8] implementation and our sequential CPU-LTR-TREE in Section IV-J.

C. Reference Implementations

We outline all of the implementations and their parameters and select good values for the parameters in Section IV-E.

GPU-NAIVE: The naive GPU algorithm uses $t \cdot |H|$ threads and 256 threads per block for the neighborhood search where t is the number of search space divisions (Section III-F).

GPU Trees: The GPU-DL-TREE and GPU-LTR-TREE use $t \cdot |H|$ threads and 256 threads per block for the neighborhood search, where t is the tree node count in the first level.

CPU-NAIVE: CPU-NAIVE is the multi-core naive CPU implementation using 64 threads/cores on PLATFORMA.

CPU Trees: CPU-DL-TREE and CPU-LTR-TREE are multi-core tree-based CPU implementations on PLATFORMA.[6]

[4]https://github.com/revanthmunugala/GDBOD

[5]All datasets except *BigCross* [21] are from the UCI Machine Learning Repository (https://archive.ics.uci.edu/).

[6]Averaging the speedup across all datasets, using 64 CPU cores compared to 1 core (sequential execution), we observe an average speedup of 19.04× for CPU-DL-TREE and an average speedup of 17.40× for CPU-LTR-TREE.

TABLE IV
SELECTED PARAMETER VALUES FOR GPU AND CPU ALGORITHMS.

Algorithm	c	Bin (b)	MinSplit (m)	Threads Per Search (t)
GPU-DL-TREE	60	7	20	Num. nodes in the first level
GPU-LTR-TREE	60	7	20	Num. nodes in the first level
CPU-DL-TREE	60	7	80	Num. nodes in the first level
CPU-LTR-TREE	60	7	40	Num. nodes in the first level
GPU-NAIVE	60	7	N/A	2
HYSORTOD	N/A	5	100	1

TABLE V
THE RESPONSE TIME (S) OF GPU-DL-TREE AND GPU-LTR-TREE WITH $m = 0$ AND $b = 7$ ON PLATFORMA USING REORDER AND NOREORDER OPTIMIZATIONS. THE HORIZONTAL DASHED LINE DEMARCATES THE COEFFICIENT OF VARIANCE THAT DETERMINES THE THRESHOLD, c.

Dataset	Coefficient of Variance (%)	NOREORDER		REORDER	
		GPU-DL-TREE	GPU-LTR-TREE	GPU-DL-TREE	GPU-LTR-TREE
Sift	20.38	954.16	442.24	3102.50	2993.02
MSD	54.60	900.90	191.42	979.69	210.85
Hepmass	57.67	134.93	92.82	104.65	100.37
BigCross	75.39	47.29	20.86	20.70	13.91
Higgs	87.76	79.90	61.77	70.14	65.69
SuSy	89.60	7.62	7.36	8.43	7.34

D. Number of Bins (b) vs. AUC Quality Score

The number of hypercubes in H is directly proportional to the number of bins (b) and has a direct effect on the quality of the produced outlierness scores. A higher value of b might lead to identifying all points as outliers whereas a lower value of b might lead to identifying all points as inliers. To strike a balance, we analyzed the AUC scores on the labeled datasets used in Cabral & Cordeiro [8] as a function of b to find an acceptable range of b with consistent quality. We observe that the AUC score remains consistent for $b \in [4, 7]$ across all datasets. We use this range to identify a good value of b for performance purposes in Section IV-E2.

E. Parameter Selection and Tuning

We describe parameter selection for our algorithms. We will use the same parameters throughout our experimental evaluation to compare the performance of our algorithms. We use the ENCDEC for hypercube construction in all parameter tuning experiments. Our algorithm has the following parameters and the selected values are summarized in Table IV.

- SELECTIVEREORDER parameter (c) - c is used in our SELECTIVEREORDER optimization to find a trade-off between the NOREORDER and REORDER optimizations.
- The number of bins (b) - The value of b is proportional to $|H|$. A lower value of b may enhance performance but may degrade outlierness score quality for each hypercube.
- *MinSplit* (m) - m is used to prune the tree and limit exhaustive tree traversals during neighborhood searches.
- Threads per hypercube (t) - t refers to the number of divisions in the search space for GPU-NAIVE.

1) SELECTIVEREORDER *Parameter (c):* As described in Section III-E, the REORDER optimization may not always yield performance gains. Table V illustrates the performance of GPU-DL-TREE and GPU-LTR-TREE with the NOREORDER and REORDER optimizations using $m = 0$ and $b = 7$. $m = 0$ refers to the complete tree whereas $b = 7$ is the maximum value of b within the acceptable range of b for AUC scores (Section IV-D). We observe that for GPU-LTR-TREE, using the NOREORDER optimization outperforms the REORDER optimization for datasets with a coefficient of variance less than 60%. Similar behavior is observed for GPU-DL-TREE except for *Hepmass* where using NOREORDER degrades performance. Based on these observations, we use $c = 60\%$ for the SELECTIVEREORDER optimization that enables REORDER only when the coefficient of variance is at least equal to the threshold coefficient of variance (c).

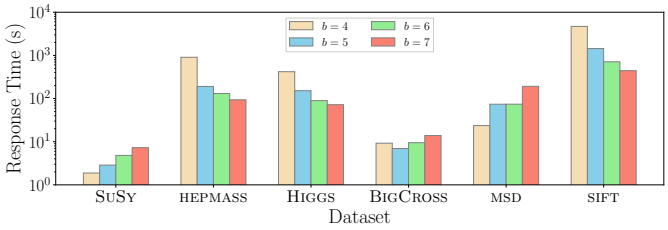
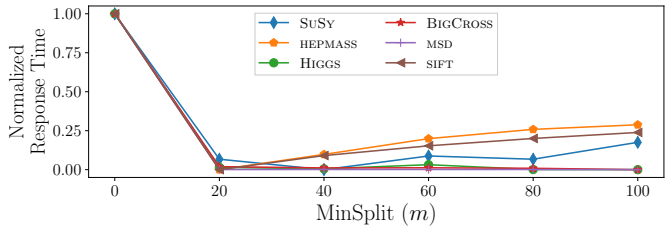


Fig. 3. The response time (s) is plotted on a log scale with $m = 0$ and $b \in [4, 7]$ for GPU-LTR-TREE executed on PLATFORMA across all datasets.

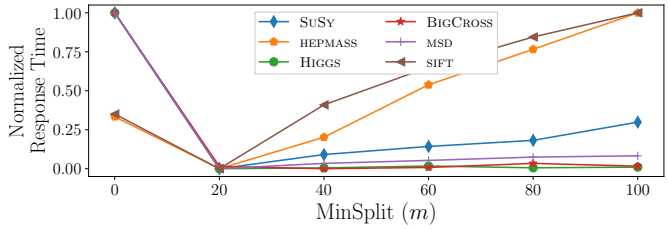
2) The number of bins (b): The bin parameter (b) affects the number of threads assigned to each hypercube as described in Section III-F during the neighborhood search. In Section IV-D, we observed that the AUC score is consistent in the range $b = 4-7$ across all datasets. Within this range, we compare the performance of each b using our GPU-DL-TREE and GPU-LTR-TREE to find a good value of b for performance purposes without affecting the quality of the outlierness score. Figure 3 plots the response times across all datasets on a log scale for GPU-LTR-TREE, with $m = 0$ and $b \in [4, 7]$. We observe that $b = 4$ yields the best performance for *SuSy* and *MSD*, $b = 5$ performs best for *BigCross*, while $b = 7$ outperforms other b parameters for *Higgs*, *Hepmass*, and *Sift*. We observe similar behavior for GPU-DL-TREE. Thus, in all that follows, we use $b = 7$ for GPU-DL-TREE and GPU-LTR-TREE.

3) MinSplit *Parameter (m):* We compare the performance of the GPU trees by varying the value of m where $m \in [0, 100]$ to identify a value that maximizes performance. Figure 4 illustrates the normalized execution time for all datasets for GPU-DL-TREE and GPU-LTR-TREE, respectively. We observe that $m = 20$ has the best performance for all the datasets for GPU-DL-TREE and GPU-LTR-TREE. In all subsequent experiments, we use $m = 20$ for both GPU trees.

4) Threads per hypercube (t): GPU-NAIVE: Recall that GPU-NAIVE scans H to find the immediate neighbors for a given hypercube h_i. The baseline uses 1 thread per hypercube, creating a total of $|H|$ threads. We increase the total number of threads by assigning t threads to perform a parallel scan for H for each $h_i \in H$, where H is partitioned into $|H|/t$ divisions (Section III-F). We observe that $t = 2$ outperforms other t values across most datasets. Higher values of t enhance parallelization but also introduce additional overhead impacting performance. Hence, we employ $t = 2$ in all future experiments using GPU-NAIVE. GPU trees: Similar to GPU-



(a) GPU-DL-Tree



(b) GPU-LTR-Tree

Fig. 4. Normalized response time on PLATFORMA, as a function of m across all datasets for (a) GPU-DL-TREE, and (b) GPU-LTR-TREE.

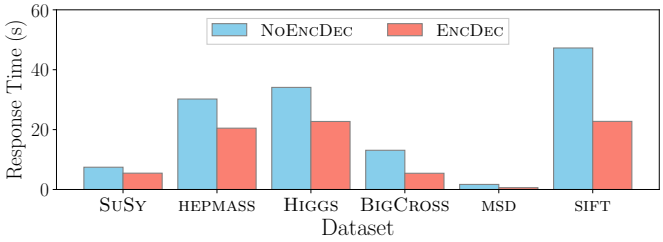


Fig. 5. The response time (s) for constructing H using our NOENCDEC and ENCDEC optimizations on the GPU on PLATFORMA.

NAIVE, multiple threads (t) are allocated for the neighborhood search of a given query hypercube. Here, the value of t is data-dependent and equals the number of nodes in the first level.

5) CPU Trees: Parameter tuning for the GPU was described above. We employ the same methodology for the CPU and select the parameters summarized in Table IV.

F. Hypercube Construction: Encoding

In Sections III-A2–III-A4, we described the construction of a distinct hypercube array H on the GPU. In this section, we compare the performance of NOENCDEC which constructs H entirely on the CPU without encoding.

Figure 5 plots the response time for hypercube construction on the GPU. We observe that the ENCDEC optimization outperforms the default, NOENCDEC, across all datasets. A speedup of $1.94\times$ for the GPU is observed on average across all datasets. This confirms that the performance gains resulting from encoding and decoding outweigh the overhead costs associated with additional operations and transfer costs between CPU and GPU in our GPU implementation. We notice a similar performance improvement in our CPU algorithm for ENCDEC, which we omit due to space limitations. We use ENCDEC in all our experimental results for both GPU and CPU algorithms in subsequent sections.

G. Comparison of Tree Methods

We observe that DL-TREE and LTR-TREE have similar performance across most datasets on the CPU, as shown in Table VI. Focusing on the GPU, we compare the performance of GPU-DL-TREE and GPU-LTR-TREE to understand the impact of locality during neighborhood searches and tree traversals, as well as how data distribution affects performance. We hypothesize that the GPU-LTR-TREE characteristic of accessing each node only once and having improved memory access during neighborhood searches will likely yield better performance over GPU-DL-TREE.

TABLE VI
THE RESPONSE TIME (S) OF GPU-DL-TREE VS. GPU-LTR-TREE USING $m = 20$ AND $b = 7$ ON PLATFORMA ALONG WITH THE PERFORMANCE OF PARALLEL CPU-DL-TREE VS. CPU-LTR-TREE USING $b = 7$ AND $m = 80$ AND $m = 40$, RESPECTIVELY.

Dataset	PLATFORMA (GPU)		PLATFORMA (CPU)	
	GPU-DL-TREE	GPU-LTR-TREE	CPU-DL-TREE	CPU-LTR-TREE
SuSy	6.49	6.47	6.75	7.78
Hepmass	93.00	89.26	523.83	561.00
Higgs	30.65	29.93	41.66	39.92
BigCross	8.14	7.94	14.93	14.73
MSD	95.05	74.16	276.03	262.72
Sift	408.52	405.16	1911.97	2246.83

TABLE VII
THE MEAN NEIGHBORHOOD DENSITY FOR DATASETS LISTED IN TABLE III USING $m = 20$ AND $b = 7$. THE MEAN NEIGHBORHOOD DENSITY AS A FRACTION OF THE TOTAL DISTINCT HYPERCUBES ARE IN PARENTHESIS.

Dataset	Dataset Size $\lvert D\rvert$	Distinct Hypercubes $\lvert H\rvert$	Mean Neighborhood Density M	Data Distribution ($M/\lvert H\rvert$)
SuSy	5,000,000	900,926	16,562	Sparse (1.838%)
Hepmass	10,500,000	10,499,723	216	Sparse (0.002%)
Higgs	11,000,000	10,680,406	283	Sparse (0.002%)
BigCross	11,620,300	1,538,368	14,674	Sparse (0.953%)
MSD	515,345	424,292	383,415	Dense (90.365%)
Sift	10,000,000	9,981,970	118	Sparse (0.001%)

Table VI shows the performance of the trees described above. Key observations are as follows: (i) GPU-LTR-TREE outperforms GPU-DL-TREE across all datasets. (ii) We observe an average speedup of $2.91\times$ for GPU-DL-TREE over CPU-DL-TREE and $3.29\times$ for GPU-LTR-TREE over CPU-LTR-TREE. Thus, GPU algorithms are significantly faster than the corresponding multi-core CPU algorithms, and the proposed locality and traversal optimizations confirm our hypothesis that GPU-LTR-TREE outperforms GPU-DL-TREE.

The performance difference between DL-TREE and LTR-TREE is minimal on both GPU and multi-core implementations primarily to the SELECTIVEREORDER optimization, which increases the likelihood of short-circuiting and the fine-grained parallelism optimization where t threads are employed to identify the neighbors of a query hypercube $h_q \in H$. However, as described in the next subsection, the GPU-LTR-TREE is more robust to differing data distributions.

H. Impact of Data Distribution on Tree Methods

Table VII shows the mean neighborhood density per hypercube. We define the mean neighborhood density per hypercube as $M = (\sum_{w_i \in W} w_i)/|H|$. We observe that *Higgs*, *BigCross*, *SuSy*, *Sift*, and *Hepmass* have sparse distributions because $M/|H| \approx 0$ whereas only *MSD* has a dense distribution.

TABLE VIII
THE RESPONSE TIME (S) OF GPU-NAIVE WITH $t = 2$ AND GPU-LTR-TREE WITH $m = 20$ AND $b = 7$ ON PLATFORMA. SPEEDUP IS THE RATIO OF THE RESPONSE TIME OF GPU-NAIVE TO GPU-LTR-TREE.

Dataset	GPU-NAIVE	GPU-LTR-TREE	Speedup
SuSy	9.47	6.47	1.46×
Hepmass	846.12	89.26	9.48×
Higgs	725.02	29.93	24.22×
BigCross	32.62	7.94	4.11×
MSD	47.45	74.16	0.64×
Sift	2,909.39	405.16	7.18×
Average Speedup			7.85×

Recall that the nodes in the GPU-DL-TREE are arranged such that the sibling nodes (same dimension nodes) are close to each other (Section III-B1). This organization is advantageous for the five datasets above that have sparse data distributions where short-circuiting is more prevalent than the dense dataset (*MSD*). For sparse distributions, GPU-DL-TREE has better locality than GPU-LTR-TREE as movement to the sibling node is preferred than moving to the child node. Even in these cases, GPU-LTR-TREE performs better than GPU-DL-TREE due to tree traversal optimizations during neighborhood searches. In summary, the performance advantage of GPU-LTR-TREE is that it is robust to a wider range of data distributions (sparse and dense) than the other tree methods.

I. GPU-NAIVE *vs.* GPU-LTR-TREE

Having established that GPU-LTR-TREE exhibits better performance than GPU-DL-TREE (Section IV-G), we compare the performance of GPU-LTR-TREE relative to GPU-NAIVE to determine how well the tree outperforms a brute force search. Table VIII illustrates the performance between GPU-NAIVE and GPU-LTR-TREE on PLATFORMA. We use $t = 2$ for GPU-NAIVE implementation and $m = 20$ and $b = 7$ for GPU-LTR-TREE. We observe that GPU-LTR-TREE outperforms GPU-NAIVE across all datasets except *MSD*. GPU-LTR-TREE exhibits an average speedup of 7.85× relative to GPU-NAIVE across all datasets. It is clear that that GPU-LTR-TREE is more effective than GPU-NAIVE for neighborhood searches on large datasets.

J. Comparison with the State-of-the-art

To make a fair comparison to HYSORTOD, we compare to our sequential CPU-LTR-TREE algorithm (without multi-threading), where we set $b = 7$ for both algorithms.[7] As outlined in Cabral & Cordeiro [8], we use the *MinSplit* parameter of $m = 100$ for HYSORTOD. For CPU-LTR-TREE we set $m = 40$. Table IX presents the total response time and speedup of CPU-LTR-TREE relative to HYSORTOD. CPU-LTR-TREE outperforms HYSORTOD across all datasets, achieving an average speedup of 18.35×. This shows that our algorithm designs are superior to HYSORTOD even for execution on a single CPU core. We also show that GPU-LTR-TREE achieves a mean speedup of 436.31× over HYSORTOD. While this is not a fair comparison, it shows that GDBOD is scalable to large datasets.

[7]The source code of HYSORTOD is at https://github.com/eug/hysortod.java.

TABLE IX
THE RESPONSE TIME (S) AND SPEEDUP OF HYSORTOD, SEQUENTIAL CPU-LTR-TREE, AND GPU-LTR-TREE USING $b = 7$, $m = 100$, $m = 40$, AND $m = 20$ RESPECTIVELY ON PLATFORMA.

Dataset	HYSORTOD	CPU-LTR-TREE	GPU-LTR-TREE
SuSy	238.71	68.61 (3.48×)	6.47 (36.89×)
Hepmass	49,918.44	14,987.76 (3.33×)	89.26 (559.25×)
Higgs	37,537.13	565.53 (66.38×)	29.93 (1254.16×)
BigCross	1,596.23	118.17 (13.51×)	7.94 (201.04×)
MSD	31,469.97	6,257.81 (5.03×)	74.16 (424.35×)
Sift	TIME_OUT	TIME_OUT (N/A)	405.16 (≥142.17×)
Average Speedup		18.35×	436.31×

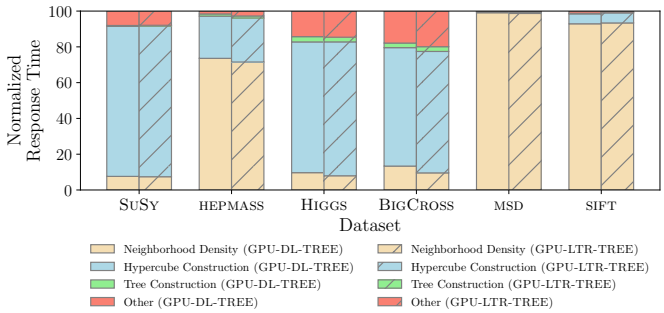


Fig. 6. Normalized response time on PLATFORMA, where the maximum time is set to 100 across all datasets for GPU-LTR-TREE using $m = 20$.

TABLE X
AVERAGED NVIDIA NSIGHT COMPUTE PROFILER STATISTICS ON PLATFORMA AND PLATFORMB. NEIGHBORHOOD SEARCH ON GPU-LTR-TREE IS COMPARED WITHOUT AND WITH ALL OPTIMIZATIONS ENABLED (SELECTIVEREORDER AND MULTITHREADS) ON THE *SuSy*, *Higgs*, AND *BigCross* DATASETS USING PARAMETERS $m = 20$ AND $b = 7$.

Platform & Optimization	Reg. Per Thread	Th. Occup.	Ach. Occup.	L1 Hits	L2 Hits	Comp. Throughput	Branch Efficiency
PLATFORMA (no opt.)	31	100%	75.69%	84.67%	71.88%	46.74%	88.36%
PLATFORMA (all opt.)	32	100%	81.85%	92.06%	92.94%	86.51%	94.35%
PLATFORMB (no opt.)	30	100%	54.17%	78.90%	58.78%	18.21%	88.36%
PLATFORMB (all opt.)	30	100%	78.91%	89.44%	84.47%	50.36%	94.35%

K. Time Distribution Analysis

Figure 6 shows the time distribution of the components of GDBOD using $m = 20$. We focus on the neighborhood density and hypercube construction components of the algorithm as they require the greatest fraction of the time. The time distribution across datasets is similar for both GPU-DL-TREE and GPU-LTR-TREE so we examine GPU-LTR-TREE. We find that the neighborhood density and hypercube construction are parallelized on the GPU and account for 48.09% and 42.98% of the total execution time on average across all datasets. The reason *Higgs*, *SuSy*, *BigCross* require a larger fraction of the total time for hypercube construction is that the neighborhood density procedure short circuits more on these datasets due to their sparse data distributions. This emphasizes the fraction of time performing hypercube construction; recall from Table VI that the response time on these datasets is lower than *MSD*, *Hepmass*, and *Sift*.

L. Datacenter vs. Consumer-grade GPUs

We compare the performance metrics of the neighborhood search on GPU-LTR-TREE using Nvidia Nsight Compute profiler with the MULTITHREADS and SELECTIVEREORDER optimizations to when these optimizations are disabled. We also compare these results between PLATFORMA and PLATFORMB to examine how the performance of GPU-LTR-TREE may differ between the two GPU models. We illustrate the results on the *SuSy*, *Higgs*, and *BigCross* datasets, as the profiler fails to collect the metrics on the other datasets.

Based on the metrics outlined in Table X, the theoretical occupancy of the kernels is 100% on both platforms indicating that there are sufficient threads to saturate the SMs and so the number of registers per thread does not limit performance. Comparing the achieved occupancy on both platforms, we find that GPU-LTR-TREE benefits from the optimizations (i.e., 75.69% increases to 81.85% on PLATFORMA and 54.17% increases to 78.91% on PLATFORMB).

Recall that good locality increases the likelihood that the data needed by a thread is already in the cache, leading to fewer expensive accesses to global memory. We find that on PLATFORMA, the L1 and L2 cache hit rates are very high, with a minimum L1 and L2 cache hit rate of 84.67% and 71.88%, respectively. Examining GPU-LTR-TREE with all optimizations enabled, we observe that the compute throughput is exceptionally high, achieving 86.51%, indicating that the implementations are able to use most of the compute capacity and this is clearly due to the high L1 and L2 cache hit rates. The Quadro RTX 5000 has lower L1 and L2 cache hit rates than the A100 because it has smaller caches (e.g., the L2 cache on the Quadro RTX 5000 is 4 MiB compared to 40 MiB on the A100). Thus, the compute throughput is lower on PLATFORMB due to memory stalls.

Recall that branching on the GPU architecture results in a loss of parallel efficiency due to warp serialization (Section III-C). We observe that the branch efficiency is exceptionally high, achieving up to 94.35% on both PLATFORMA and PLATFORMB. This indicates that, in most cases, the threads within the warp follow a uniform execution path mitigating the branch divergence during neighborhood searches.

In summary, the experiments illustrate that the traversal and locality properties of GPU-LTR-TREE are well optimized.

V. DISCUSSION & CONCLUSIONS

In this paper, we present GDBOD and compared the performance of two tree-based variants of the algorithm and found that the locality and traversal properties of GPU-LTR-TREE outperformed GPU-DL-TREE. In particular, tree traversals on the GPU typically yield uncoalesced memory accesses and thus poor locality, and instruction serialization due to branching as a consequence of the SIMT architecture. Despite these challenges with our proposed optimizations on the GPU, we achieve a peak compute throughput of 86.51%, along with 92.06% L1 cache hits and 92.94% L2 cache hits on real-world datasets. For systems without GPUs, our sequential CPU algorithm achieves an average speedup of 18.35× when compared to HYSORTOD. Under practical time constraints, the GPU algorithm allows for scaling to much larger datasets or those with higher dimensionality than prior work.

REFERENCES

[1] V. Chandola, A. Banerjee, and V. Kumar, "Anomaly detection: A survey," *ACM computing surveys*, vol. 41, no. 3, pp. 1–58, 2009.

[2] H. Wang, M. J. Bah, and M. Hammad, "Progress in Outlier Detection Techniques: A Survey," *IEEE Access*, vol. 7, pp. 107 964–108 000, 2019.

[3] G. H. Orair, C. H. Teixeira, W. Meira Jr, Y. Wang, and S. Parthasarathy, "Distance-based Outlier Detection: Consolidation and Renewed Bearing," *Proc. of the VLDB Endowment*, vol. 3, pp. 1469–1480, 2010.

[4] M. Goldstein and S. Uchida, "A comparative evaluation of unsupervised anomaly detection algorithms for multivariate data," *PloS one*, vol. 11, no. 4, p. e0152173, 2016.

[5] E. Schubert, A. Zimek, and H.-P. Kriegel, "Local outlier detection reconsidered: a generalized view on locality with applications to spatial, video, and network outlier detection," *Data mining and knowledge discovery*, vol. 28, pp. 190–237, 2014.

[6] E. M. Knorr and R. T. Ng, "A unified notion of outliers: Properties and computation." in *KDD*, vol. 97, 1997, pp. 219–222.

[7] G. O. Campos, A. Zimek, J. Sander, R. J. Campello, B. Micenková, E. Schubert, I. Assent, and M. E. Houle, "On the evaluation of unsupervised outlier detection: measures, datasets, and an empirical study," *Data mining and knowledge discovery*, vol. 30, no. 4, pp. 891–927, 2016.

[8] E. F. Cabral and R. L. Cordeiro, "Fast and scalable outlier detection with sorted hypercubes," in *Proc.of the 29th ACM Intl. Conf. on Information & Knowledge Management*, 2020, pp. 95–104.

[9] F. Angiulli, S. Basta, S. Lodi, and C. Sartori, "GPU strategies for distance-based outlier detection," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 11, pp. 3256–3268, 2016.

[10] T. Matsumoto and E. Hung, "Accelerating outlier detection with uncertain data using graphics processors," in *Pacific-Asia Conference on Knowledge Discovery and Data Mining*. Springer, 2012, pp. 169–180.

[11] C. HewaNadungodage, Y. Xia, and J. J. Lee, "GPU-accelerated outlier detection for continuous data streams," in *IEEE Intl. Parallel and Distributed Processing Symp.*, 2016, pp. 1133–1142.

[12] E. M. Knox and R. T. Ng, "Algorithms for mining distance based outliers in large datasets," in *Proc. of the Intl. Conf. on Very Large Data Bases*, 1998, pp. 392–403.

[13] S. Papadimitriou, H. Kitagawa, P. B. Gibbons, and C. Faloutsos, "Loci: Fast outlier detection using the local correlation integral," in *Proc. of the IEEE Intl. Conf. on Data Engineering*, 2003, pp. 315–326.

[14] M. Nam, J. Kim, and B. Nam, "Parallel tree traversal for nearest neighbor query on the GPU," in *45th Intl. Conf. on Parallel Processing*, 2016, pp. 113–122.

[15] J. Kim, S.-G. Kim, and B. Nam, "Parallel multi-dimensional range query processing with R-trees on GPU," *Journal of Parallel and Distributed Computing*, vol. 73, no. 8, pp. 1195–1207, 2013.

[16] M. Goldfarb, Y. Jo, and M. Kulkarni, "General transformations for GPU execution of tree traversals," in *Proc. of the Intl. Conf. on High Performance Computing, Networking, Storage and Analysis*, 2013, pp. 1–12.

[17] F. Zhang, P. Di, H. Zhou, X. Liao, and J. Xue, "RegTT: Accelerating tree traversals on GPUs by exploiting regularities," in *Intl. Conf. on Parallel Processing*, 2016, pp. 562–571.

[18] D. Merrill, M. Garland, and A. Grimshaw, "High-performance and scalable GPU graph traversal," *ACM Transactions on Parallel Computing*, vol. 1, no. 2, pp. 1–30, 2015.

[19] M. Gowanlock, B. Gallet, and B. Donnelly, "Optimization and Comparison of Coordinate-and Metric-Based Indexes on GPUs for Distance Similarity Searches," in *Intl. Conf. on Computational Science*. Springer, 2023, pp. 357–364.

[20] M. Gowanlock and B. Karsin, "GPU-accelerated Similarity Self-join for multi-dimensional Data," in *Proc. of the Intl. Workshop on Data Management on New Hardware*, 2019, pp. 1–9.

[21] M. R. Ackermann, M. Märtens, C. Raupach, K. Swierkot, C. Lammersen, and C. Sohler, "Streamkm++: A clustering algorithm for data streams," *ACM J. Exp. Algorithmics*, vol. 17, may 2012.