GPU-Accelerated Similarity Self-Join for Multi-Dimensional Data

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

Ben Karsin
Department of Computer Science
Université libre de Bruxelles
Brussels, Belgium
Benjamin.Karsin@ulb.ac.be

ABSTRACT

The similarity self-join finds all objects in a dataset that are within a search distance, ϵ, of each other. As such, the self-join is a building block of many algorithms. In high dimensions, indexing structures become increasingly ineffective at pruning the search, making the self-join challenging to compute efficiently. We advance a GPU-accelerated self-join algorithm targeted towards high dimensional data. The massive parallelism afforded by the GPU and high aggregate memory bandwidth makes the architecture well-suited for data-intensive workloads. We leverage a grid-based GPU-tailored index to perform range queries, and propose the following optimizations: (i) a trade-off between candidate set filtering and index search overhead by exploiting properties of the index; (ii) reordering the data based on variance in each dimension to improve the filtering power of the index; and (iii) a pruning method for reducing the number of expensive distance calculations. Our algorithm generally outperforms a parallel CPU state-of-the-art approach.

CCS CONCEPTS

• **Information systems** → *Data management systems*; • **Computing methodologies** → *Parallel algorithms*; • **Computer systems organization** → *Single instruction, multiple data.*

KEYWORDS

GPGPU, High-dimensional data, Index structure, In-memory database, Query optimization, Self-join

ACM Reference Format:
Michael Gowanlock and Ben Karsin. 2019. GPU-Accelerated Similarity Self-Join for Multi-Dimensional Data. In *International Workshop on Data Management on New Hardware (DaMoN'19), July 1, 2019, Amsterdam, Netherlands.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3329785.3329920

1 INTRODUCTION

The similarity self-join is a building block of several algorithms [3, 5, 8, 22, 36], and is fundamental to many established methods [12, 15]. We focus on the distance similarity self-join that finds all objects within a Euclidean distance of each other. Self-join research addresses either low [16] or high [4, 20, 39] dimensionality. Typically, indexes (e.g., R-trees [29], kd-trees [10], and quad trees [23]) are used to reduce the number of distance comparisons when performing neighborhood searches. They accomplish this by eliminating a large number of points before performing distance comparisons by pruning the search space. In low dimensionality (low-D), the data points (or feature vectors) are often more frequently co-located; therefore, there are often more neighbors on average in comparison to high dimensionality (high-D) [32]. The large number of resulting candidate points requires in a large number of distance calculations, which become a performance bottleneck. However, in high-D, index searches become increasingly exhaustive due to the well-known curse of dimensionality [9, 21, 34, 45] that requires searching a large fraction of the dataset. Thus, there is both more index search overhead, and distance calculations that are needed to refine potential candidates within ϵ.

Graphics processing units (GPUs) obtain high computational throughput through massive parallelism and high aggregate memory bandwidth. The self-join is well-suited for the GPU because it requires many independent distance calculations. Therefore, the GPU can be exploited to process increasingly exhaustive searches necessitated by high dimensional data. In this context, we propose several optimizations that improve performance on the GPU, and make the following contributions:

- We exploit trading index filtering power for decreased search cost to optimize high-D index searches.
- We improve the filtering power of the index by reordering the data in each dimension using statistical properties.
- We mitigate the cost of reducing index filtering power by proposing a technique that prunes the candidate set by comparing points based on an un-indexed dimension.
- We show that our algorithm is resilient to the worst-case data distribution.
- We evaluate the performance of our self-join algorithm on a range of synthetic and real-world datasets and demonstrate that it outperforms a state-of-the-art algorithm.

The paper is organized as follows. Section 2 provides background material, Section 3 formalizes the problem, and outlines leveraged work, Section 4 presents novel self-join optimizations, Section 5 evaluates our approach, and finally, we conclude the paper and discuss future work in Section 6.

DaMoN'19, July 1, 2019, Amsterdam, Netherlands
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6801-8/19/07…$15.00
https://doi.org/10.1145/3329785.3329920

2 BACKGROUND

In this section, we outline related work on similarity joins in shared and distributed memory, and GPU indexing techniques. We begin by outlining the motivation for using the GPU for database operations.

2.1 Motivation: Using the GPU for Fundamental Database Operations

The proliferation of GPUs to solve problems in many fields of computer science, including within the database community, has been motivated by several facets of the architecture. The memory bandwidth on the GPU is much greater than the CPU [1], which makes the GPU an attractive architecture for solving data-intensive problems. Also, the GPU has many cores, which can be most efficiently utilized for high throughput applications that are common to many classes of database queries. Furthermore, GPUs have been noted for having a greater energy efficiency than the CPU for many applications [42], and have been employed for their relatively low monetary cost per unit metric (e.g., floating point operations per second) [30, 44]. Due to the characteristics outlined above, the most powerful supercomputers rely on GPUs. At the time of writing, five of the top ten supercomputers in the world use GPUs (Top500 November 2018 listing [2]). The architectural features of the GPU are well-suited to many parallel database applications; therefore, we advocate for exploiting the GPU to improve the performance of the distance similarity self-join.

2.2 Similarity-Joins and the State-of-the-art

The similarity-join is a well-studied problem [6, 8, 12, 14, 20, 33]. Here, we discuss those works that address high-D data. GESS [20] assigns feature vectors to hypercubes, and then performs an intersection query on these hypercubes to compute the similarity join. The method relies on data replication and duplication removal from the result set. LSS [39] utilizes the GPU, and transforms the similarity join into a sort-and-search problem. Interval searches are needed, and the authors use space filling curves to reduce interval size and search overhead. The Super-EGO algorithm [33] has been shown to be effective for similarity-joins on both low-D and high-D data. The algorithm uses the "epsilon grid order" [13] method. It uses a non-materialized grid to find nearby points that may be within the search distance. Then, based on a query point's cell and nearby cells, the algorithm prunes the search for points by filtering on n-dimensional coordinates. Unlike previous work [13], Super-EGO exploits statistical properties of the data. In [33], Super-EGO outperforms GESS [20], and LSS [39], so we compare our work to Super-EGO.

2.3 GPU Self-Join on Low-Dimensional Data

Gowanlock and Karsin [27] studied the self-join problem on the GPU for low-D data using a grid-based index, and demonstrated that between 2 and 6 dimensions, the self-join outperforms both canonical search-and-refine and state-of-the-art approaches (i.e., Super-EGO). They show that index search overhead increases exponentially with dimensionality, and they limit their work to low-D data. In this work, we use a similar indexing structure, but we propose optimizations for high-D self-joins.

2.4 Indexing on the GPU

There are two major indexing strategies for the GPU: (*i*) index-trees, similar to those that have been shown to provide good performance on the CPU, such as the R-tree [29, 41]; or (*ii*) non-hierarchical indexes, such as grids or binning. Several works propose efficient indexes for points or other objects on the GPU [15, 26, 34, 35, 37, 46].

Kim et al. [34] designed an R-tree for the GPU to optimize index searches that avoids many of the drawbacks of executing tree traversals on the GPU. A major drawback of tree traversals is that their irregular instructions cause thread divergence. This divergence reduces the parallel efficiency on the GPU due to the single instruction multiple thread (SIMT) architecture [40]. Later, the same research group presented a hybrid R-tree indexing approach [35] that splits the R-tree between the CPU and GPU by assigning parts of the algorithm with more regular and irregular instruction flows to the GPU and CPU, respectively. The reduction in irregular instructions allows the GPU to achieve better performance. Likewise, the approach used in [37] for computing range queries on moving objects using the CPU and GPU addressed many of the idiosyncrasies of the GPU's architecture. Therefore, the design space for efficient GPU indexing techniques is large, so efficient GPU-only and CPU/GPU indexing techniques remain largely an open problem with little consensus on the best indexing approach.

2.5 Distributed-Memory Similarity Joins

High-D self-joins are expensive for even moderate dataset sizes. While the literature above focus on scaling *up* the self-join, several other works scale *out* across nodes in a cluster. A MapReduce [19] self-join [24] reduces data duplication compared to previous work [43], by using "dimension groups", where they perform the self-join on subsets of the data dimensions first, and then union these subsets to obtain the final result. Similarly, the works of [17] and [18] use MapReduce for similarity joins, and employ sampling-based techniques for data partitioning to achieve good load balancing. Distributed-memory works are not directly relevant to GPU self-joins, as exploiting the GPU requires considering a much smaller degree of task granularity.

3 PROBLEM OUTLINE & PREVIOUS INSIGHTS

3.1 Problem Statement

Let D be a database of points (or feature vectors). Each point in the database is denoted as p_i, where $i = 1, 2, \ldots, |D|$. Each $p_i \in D$ has coordinates in n dimensions, where each coordinate is denoted as x_j where $j = 1, 2, \ldots, n$. Thus, the coordinates of point p_i are denoted as: $p_i = (x_1, x_2, \ldots, x_n)$. We refer to the x_j-coordinate value of point p_i as $p_i(x_j)$. As with most prior related work (Section 2), we focus on the Euclidean distance. We find all pairs of points that are within a distance ϵ of each other. We say that points $a \in D$ and $b \in D$ are within the ϵ distance when $dist(a, b) \leq \epsilon$, where $dist(a, b) = \sqrt{\sum_{j=1}^{n}(a(x_j) - b(x_j))^2}$. Thus, the result set contains tuples $(a \in D, b \in D)$, where a and b are within ϵ of each other.

We assume that the dataset, result, and working memory do not exceed main memory capacity. However, we accommodate result sets that exceed the GPU's global memory capacity.

The similarity self-join is a special case of the similarity join. If we let E be a set of entry points in an index (defined similarly to the definition above) and Q be a set of query points, the similarity join finds all points in Q within the ϵ distance of E, i.e., $Q \ltimes_\epsilon E$. In contrast, the self-join is simply $E \bowtie_\epsilon E$. Thus, the self-join is relevant to the similarity-join problem as well.

The worst-case complexity of the self-join, $O(|D|^2)$, can be simply implemented as a nested loop join [31]. However, as discussed in Section 2, indexes can be used to reduce the quadratic complexity by reducing the number of point comparisons by pruning the search.

3.2 Leveraging Previous Insights

In Section 4, we outline our novel methods for performing the self-join in high-D. However, we leverage several optimizations from the literature that are relevant to the self-join. In particular, we use the grid-based GPU index presented by Gowanlock & Karsin [27], that builds on prior work [28]. These papers also advanced a batching scheme, which we use to process self-join result sets that may exceed the GPU's global memory capacity. We briefly describe the batching and indexing techniques that we reuse, and note that we cannot directly use the low-D methods [27] for high-D self-joins.

Grid-Based Indexing on the GPU – We utilize a grid index for computing the self-join. As mentioned in Section 2, the state-of-the-art join algorithm for high-D data, Super-EGO [33], also uses a grid-based technique for efficiently computing the self-join. We refer the reader to the work of Gowanlock et al. [28] for an in-depth description of the index, which the authors used in 2-D for clustering with DBSCAN [22]. A major difference between the indexing scheme used in this work and that of Gowanlock et al. [28] is that we do not index empty cells, as the space complexity would be intractable for high-D (as also discussed in [27]).

The GPU grid index from Gowanlock et al. [28] is constructed as follows. On the host, the data points, D, are sorted into unit-length bins in each dimension. This ensures that data points near each other in the n-dimensional space are near each other in memory. Each grid cell is of length ϵ, which ensures that for a given point, only the adjacent cells need to be searched to find points that are within the ϵ distance. This bounds the search on the GPU to regularize the instruction flow. For demonstrative purposes and without loss of generality, we assume a grid with edges starting at 0 in each dimension, and assign points to cells by simply computing the cell's n-dimensional coordinates from the point's (p_i) coordinates as follows: $(x_1/\epsilon, x_2/\epsilon, \ldots, x_n/\epsilon)$. The points are not stored within the grid structure, rather, the points belonging to a grid cell are stored in a lookup array that each grid cell references when finding the points contained within. This minimizes the memory needed to store the points within a grid cell. Lastly, since we only store non-empty cells, we create a lookup array that stores the linearized ids of the non-empty grid cells. As shown by Gowanlock and Karsin [27], the storage requirements simplify to the size of the dataset, $O(|D|)$. This compact index structure allows more space on the GPU to be allocated for other purposes, such as larger input and result set sizes.

Figure 1 shows an example 2-D grid. The non-empty gray cells with linearized cell ids are shown. Consider a point in cell 24. To find all of its neighbors within ϵ, it needs to search the adjacent cells

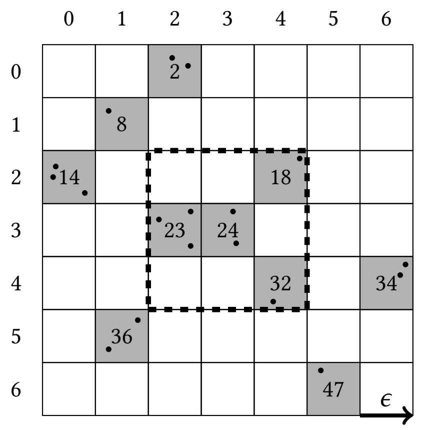


Figure 1: Example of searching the grid index in 2-D. The non-empty cells are shaded. Numbers refer to linearized cell ids.

(and its origin cell), which are encompassed by the black dashed line. In n dimensions, there are 3^n cells to search. However, the points in the cells are not guaranteed to be within ϵ, so distance calculations between the query point and all points in cells 18, 23, 24, and 32 are needed to determine which are within ϵ.

The self-join is executed on the GPU with a *kernel* that uses $|D|$ threads. Each thread is assigned a point and finds all neighbors within the ϵ distance. The threads write the result to a buffer as key/value pairs, where the key is a point and the value is a point within ϵ of the key. After all threads have completed finding their respective neighbors, the key/value pairs are sorted on the GPU, and returned to the host.

Bounding the search to neighboring cells using the grid reduces thread divergence, which can degrade GPU performance [40]. This is because all searches require examining the same number of cells regardless of the point, and the cells are traversed in the same manner. In contrast, indexes that are constructed based on the data distribution, such as R-trees, would require irregular searches (threads take different branches during tree traversals), which would increase thread divergence in a warp.

Batching Scheme – An efficient batching scheme is needed to incrementally compute the self-join to accommodate result sets that exceed the GPU's global memory capacity. We employ the method from [28], and provide a summary of their work. First, a kernel is executed that finds all of the neighbors within ϵ for a fraction of the points in the dataset, which estimates the total result set size. This kernel invocation takes negligible time in comparison to the total time needed to execute the self-join, as only a fraction of points are searched. The number of batches, n_b, are computed based on a batch size, b_s, and the estimated total result size.

The batching scheme allows for overlap of data transfers to and from the GPU, GPU computation, and host-side operations. It is preferable to overlap these components of the algorithm to maximize concurrent resource utilization. Thus, we use a minimum of 3 CUDA streams, and hence batches ($n_b \geq 3$). We allocate 3 *pinned memory* buffers on the host, as they are needed for asynchronous data transfers [40]. For result set sizes that exceed 3×10^8, we set a batch size of $b_s = 10^8$ (the total neighbors found within ϵ of each point). Thus, each stream has a buffer of size $b_s = 10^8$.

4 HIGH-D SELF-JOIN OPTIMIZATIONS

In this section, we introduce our optimizations that are designed to improve high-dimensional self-join performance.

4.1 Index Selectivity

In high-D, there are fewer co-located neighbors because, as the hypervolume increases, the distance between objects increases [32]. However, with increasing dimension, index filtering power decreases and search performance degrades. There is a trade-off between index filtering power and search overhead: reducing search overhead results in an index with less filtering power, yielding larger candidate set sizes that are filtered using distance calculations.

The GPU is a suitable architecture for making a trade-off between filtering power and search overhead, as the GPU is designed to achieve high computational throughput and thus excels at computing the distances between points in parallel. Therefore, to avoid the overheads associated with searches in higher dimensions, we use a less rigorous index search at the cost of increased filtering overhead. To illustrate why this trade-off is important in the context of the grid indexing scheme, the number of adjacent cells required to check is 3^n; in 2-D, this is only 9 cells, but in 6-D, this is 729 cells. We decrease the filtering power and search overhead by indexing only k dimensions of the n-dimensional points, where $2 \leq k < n$, thus projecting the points into k dimensions. To resolve whether points are within ϵ of the query point, we compute the Euclidean distance in all n dimensions, and thus obtain the correct result. Since we index in fewer than n dimensions, each cell has $n - k$ unconstrained dimensions, resulting in less filtering power.

4.2 Dimensionality Reordering by Variance

Index searches are increasingly exhaustive and more expensive in higher dimensions. The statistical properties of high-D feature vectors can be exploited to improve the filtering power of the index to prune the search space and eliminate points that are not within ϵ (e.g., see [33] in related work, Section 2). The dimensions of the data that have the greatest variance should improve the pruning power of index searches and, since we may not index all dimensions, it is important to select dimensions that optimize the pruning power. Otherwise, if we select the first k dimensions, we may inadvertently index on dimensions that yield minimal pruning power.

Figure 2(a) shows an example dataset of 10 points in 6 dimensions generated in the range [0,1]. We can see that the first two dimensions have a low degree of variance. Thus, if we index $k = 3$ dimensions (and not all $n = 6$), we will have a low amount of index filtering power due to low variance in the first two dimensions. Assuming that the grid cells are of length $\epsilon = 0.2$, we find that dimensions 1 and 2 will only produce a single cell in their dimensions (denoted by the shaded regions), and thus will not reduce the number of points within ϵ. Selecting dimensions with the greatest variance improves the filtering power (i.e., dimensions 5, 3, and 6 in Figure 2(a)). If we reorder the data by decreasing variance, then we obtain Figure 2(b). Now, each of the first 3 dimensions spans 5 grid cells (assuming $\epsilon = 0.2$), resulting in fewer points when searching.

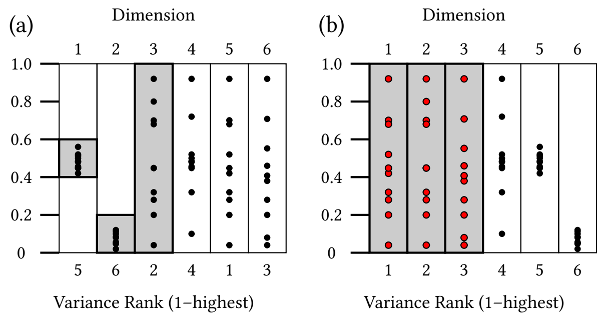


Figure 2: Dimensionality reordering by variance on a 6-D dataset having $|D| = 10$ indexing $k = 3$ dimensions. (a) input dataset; (b) reordering the point coordinates from largest to smallest variance in each dimension. Red points denote those used to index $k = 3$ dimensions with high variance. Shaded cells denote indexed area.

We note that in Figure 2, it seems like the *number* of cells should be maximized and not the *variance*. While data with high variance will tend to produce more cells, it is possible to have many cells in a dimension with low variance (e.g., one point per cell, and the remaining points in a single cell, as in dimension 4 in Figure 2(b)).

To re-order the dimensions by their variance, we use a sample of 1% of $|D|$ and estimate the variance in each dimension. Then, we reorder the coordinate values in each dimension of $p_i \in D$, such that the values are in descending order from highest to lowest variance. Thus, when we index the first k dimensions (Section 4.1), they potentially have greater filtering power than the initial input dataset. Reordering dimensions does not impact the correctness of the result, as we are simply swapping the coordinate values of the points. This requires $O(|D|n)$ work, which is negligible compared to performing the self-join. We denote the optimization that reorders the data by variance in each dimension as REORDER. If we index $k < n$ dimensions, but do not use REORDER, we simply index the first k dimensions of the input dataset.

4.3 Searching on an Un-indexed Dimension

By indexing only $k < n$ dimensions, we reduce the indexing overhead by reducing the number of grid cells, which is exponential with k. However, this comes at the cost of reduced filtering power, resulting in more distance calculations. In this section we introduce a technique of searching on an un-indexed dimension to further reduce the number of necessary distance calculations. Consider an input set with n dimensions that is indexed on $k < n$ dimensions using the indexing scheme presented in Section 3.2. For a given point p in cell C_a and neighbor cell C_b, we compare p to each point $q \in C_b$ to determine if p and q are within a distance ϵ of each other. Since we have indexed k dimensions, the points contained in C_b are only filtered by these k dimensions. Thus, if we consider dimension u that is *not* indexed, each point in C_b can have any value in this dimension. Currently, we must perform a distance comparison on all $q \in C_b$, which includes such points that may be

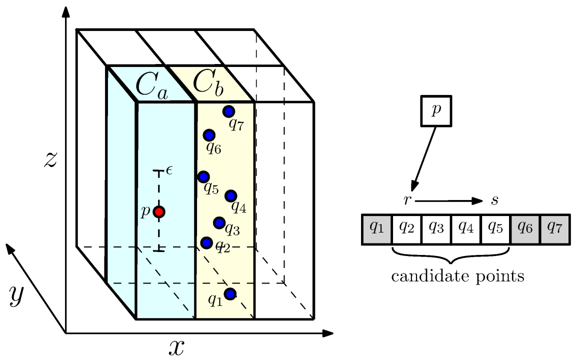


Figure 3: Example of the SORTIDU optimization. We first sort points within each cell by z-coordinate. We then search query point p into the list of C_b to find r and scan and compare points until s is reached.

very distant from p in dimension u (i.e., $|p(u) - q(u)| > \epsilon$). Therefore, we propose an optimization called SORTIDU to only compare p with $q \in C_b$ if they are within ϵ distance along the u-coordinate. We accomplish this by first sorting the points within each cell by increasing u-coordinate. When comparing p with all $q \in C_b$, we first search $p(u)$ into the points in C_b to find the point r with the smallest u-coordinate that is still within ϵ of p (i.e, $|p(u)-r(u)| \leq \epsilon$). We then scan points in C_b by increasing u coordinate until we reach point s with more than the ϵ distance in the u-coordinate (i.e., $|p(u) - s(u)| > \epsilon$). Figure 3 illustrates an example of the SORTIDU optimization. In this example, the z-axis is not indexed and we use the SORTIDU to reduce the number of candidate points we have to consider, from $q_1, q_2, \ldots, q_7$ (7 points) to $q_2, \ldots, q_5$ (4 points). We note that we only perform this optimization on one un-indexed dimension and all other un-indexed dimensions remain unfiltered.

If every point in C_b is within ϵ from p, then SORTIDU provides no performance improvement. However, for reasonably small ϵ values, this can significantly reduce the number of candidate points. This comes at the cost of sorting and searching. If we consider that cell C_b has $|C_b|$ points, we must perform $|C_b|$ distance calculations without the SORTIDU optimization. However, SORTIDU reduces this to $(\log |C_b| + m)$ calculations, where m is the number of points in C_b with u-coordinate within ϵ of $p(u)$. While SORTIDU requires that we sort points within each cell, we only have to sort once for all point evaluations. We can apply the SORTIDU optimization even when we index all dimensions (i.e., $k = n$), where we sort each cell by one of the indexed dimensions. But, we expect more significant benefits when we apply SORTIDU to an un-indexed dimension.

4.4 Short Circuiting the Distance Calculation

The cost of the distance calculation increases with dimensionality. We incrementally compute the distance, and if the partial sum exceeds ϵ before the entire distance is calculated, we stop the calculation early. Short-circuiting has been used in other works, such as SUPER-EGO [33]. We denote this optimization as SHORTC.

Algorithm 1 GPU-JOIN Algorithm

```
1:  procedure GPU-JOIN($\epsilon$, $n$, $k$, $b_s$)
2:      $D \leftarrow$ importData()
3:      $D \leftarrow$ reorderVariance($D$)
4:      $G \leftarrow$ constructIndex($D$, $k$)
5:      $n_b \leftarrow$ computeNumBatches($b_s$)
6:      result $\leftarrow \emptyset$
7:      for $i \in 1,\ldots,n_b$ do
8:          kernelResult[i] $\leftarrow$ selfJoinKernel($D$, $G$, $n$, $k$, $\epsilon$)
9:          result $\leftarrow$ result $\cup$ constructNeighborTable(kernelResult[i])
10:     return

11: procedure SELFJOINKERNEL($D$, $G$, $n$, $k$, $\epsilon$)
12:     resultSet $\leftarrow \emptyset$
13:     gid $\leftarrow$ getGlobalId()
14:     point $\leftarrow$ getPoint(gid, $D$)
15:     adjCells $\leftarrow$ getAdjCells($G$, $k$, point)
16:     for cell $\in$ adjCells.min,. . . ,adjCells.max do
17:         pntResult $\leftarrow$ pntResult $\cup$ calcDistancePts(point, cell, $n$, $\epsilon$)
18:     resultSet $\leftarrow$ resultSet $\cup$ pntResult
19:     return resultSet
```

4.5 Outline of the Algorithm

Algorithm 1 begins by re-ordering the input set, D, by variance, if REORDER is enabled (line 3). Next, the index is computed using the dataset and the number of indexed dimensions, k (line 4). Then, the number of batches, n_b, to be executed are computed using the batch size, b_s (line 5). The algorithm then loops over all of the batches (line 7) and executes them on the GPU (line 8, detailed below). The result of each batch is stored as key/value pairs, where the key is a query point and the value is a point within the ϵ distance (i.e., for each query point p_i there may be multiple result pairs (p_i, p_j)). Since the keys are often redundant (multiple points are within ϵ of a given point), they are stored without redundant information using constructNeighborTable, and stored in the final result (line 9).

Each batch is executed by the SELFJOINKERNEL GPU kernel. First, the result set for the batch is initialized (line 12), the global id of the thread is obtained (line 13), and the point, $p_i \in D$ is computed as a function of the global id (line 14). Next, all of the adjacent non-empty cells are computed from G (the index), k, and the point (line 15). The algorithm loops over each neighbor cell (line 16), and computes the distance between the query point and all of the points within the neighbor cell to determine if they are within ϵ (line 17). After the points within ϵ of the query point have been added to the result set, the kernel returns (lines 18–19). Function calcDistancePts differs when using SORTIDU or SHORTC.

Parts of Algorithm 1 are pipelined. The loop on line 7 is executed in parallel. For each batch, four components may be executed in parallel: (i) the kernel parameters relevant to the batch are sent to the GPU; (ii) the GPU kernel, SELFJOINKERNEL, computes the self-join result for the batch; (iii) the result is transferred back to the host; and, (iv) finally the neighbor table is constructed. Up to four concurrent tasks can be executed, where overlapping data transfers and computation hides communication overhead.

5 EXPERIMENTAL EVALUATION

5.1 Datasets

We use real and synthetic datasets. We use some of the real-world datasets used to evaluate Super-Ego, denoted as SUPER-EGO [33] (*ColorHist* and *LayoutHist*). We normalize all datasets in the range

[0,1] as required by Super-EGO. Real-world datasets were obtained from UCI ML repository [38]. Datasets are as follows:

- Color Histogram, *ColorHist*– 32-D image features, and 68,040 points.
- Layout Histogram, *LayoutHist*– 32-D image features and 66,616 points.
- Supersymmetry Particles, *SuSy*– 18-D kinematic properties of 5 million particles from the Large Hadron Collider. Used for classification [7].
- Song Prediction Dataset, *Songs*– 90-D extracted features of songs, with 415,345 points. Used for classification [11].

We use real-world datasets to evaluate performance of high-D self-joins. However, the similarity joins rely on statistical techniques for improving index efficacy. The *worst case* scenario for our algorithm is when there is low variance across all dimensions, reducing the impact of dimensionality reordering (Section 4.2). To evaluate performance on such worst-case inputs, we generate synthetic datasets with an exponential distribution with $\lambda = 40$. We generate 16, 32, and 64-dimensional synthetic datasets, with coordinates in [0,1] and with $|D| = 2 \times 10^6$, denoted *Syn-* (*Syn16D2M*, *Syn32D2M*, and *Syn64D2M*). The datasets use an exponential distribution to ensure that in high-D, there are some neighbors within ϵ.

5.2 Experimental Methodology

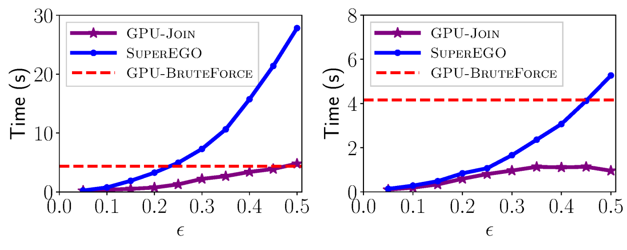
The GPU code is written in CUDA and executed on an NVIDIA GP100 GPU with 16 GiB of global memory. The C/C++ host code is compiled with the GNU compiler (v. 5.4.0) and O3 optimization flag. The platform that executes all experiments has 2× E5-2620 v4 2.1 GHz CPUs, with 16 total physical cores. Our self-join CUDA kernel uses 256 threads per block, and uses 32-bit floats for consistency with Super-EGO. We exclude the time to load the dataset and construct the index. We include the time to execute the self-join, store the result set on the host and other host-side operations, and perform dimensionality reordering. Thus, we include the response time of components used in other works to make a fair comparison.

We perform experiments across datasets and ϵ values such that we do not have too many (e.g., $|D|^2$) or too few (e.g., 0) total results. Thus, the values of ϵ should represent values that are pragmatically useful to a user of the self-join algorithm. We define the selectivity of the self-join as $S_D = (|R| - |D|)/|D|$, where $|R|$ is the total result set size. This yields the average number of points within ϵ, excluding a point, p_a, finding itself (i.e., a result tuple: $(p_a \in D, p_a \in D)$). We report the selectivity in our plots so that our results can be reproduced and to demonstrate that the respective experimental scenario is meaningful. Our experiments cover a range of ϵ values that include those used by Kalashnikov [33] to evaluate Super-EGO.
GPU-BruteForce– We compare our approach to a $O(|D|^2)$ brute force algorithm. We simply assign one thread per query point which then computes the distance between the query point and all other points in D. However, unlike the GPU-Join kernel, we do not return the result, and instead simply count the total number of points within ϵ. Thus, we only execute a single kernel invocation, which yields a lower bound on the brute force response time. Since all points are compared to each other, performance is roughly independent of ϵ; therefore, in our results we show the brute force time corresponding to a single ϵ value (the median ϵ in the plots).

Reference Implementation (Super-EGO) – Super-EGO [33] performs fast self-joins on multidimensional data and has been shown to outperform other algorithms on low-D and high-D data. We use a multi-threaded implementation of Super-EGO, using 16 threads on 16 physical cores (the number of cores on our platform). We normalize the datasets in the range [0,1] in each dimension, as needed by the algorithm. We compute the total time using the time to ego-sort and join on 32-bit floats and exclude the other components (e.g., loading the dataset and indexing).

5.3 Results

5.3.1 Performance on Small Datasets. We compare GPU-Join to Super-EGO, for $n = 32$ and varying ϵ, using some of the same datasets as those used by Kalashnikov [33]. We do not use any optimizations except indexing $k < n$ dimensions (Section 4.1). We discuss the selection of k in an upcoming section. Figure 4 plots response time vs. ϵ on *ColorHist* and *LayoutHist*. We find that across all datasets, GPU-Join tends to outperform Super-EGO (which uses 16 cores/threads). In Figure 4(b) we see that GPU-Join and Super-EGO have nearly identical performance when $\epsilon \leq 0.15$, but diverge when $\epsilon > 0.15$. The reason the performance of GPU-Join does not degrade significantly with ϵ is because these datasets are relatively small and there is not enough work to fully saturate the GPU's resources. Thus, as ϵ increases, the response time does not increase in the same manner as Super-EGO. We observe that GPU-Join outperforms GPU-BruteForce across almost all values of ϵ, except when $\epsilon = 0.5$ on *ColorHist* (Figure 4(a)). When $\epsilon = 0.5$, the selectivity is very high, so for each query point, GPU-Join needs to compute the distance to a large fraction of the other points in the dataset. Interestingly, GPU-BruteForce outperforms Super-EGO on some of the larger ϵ values.

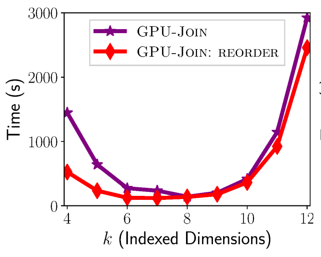


(a) *ColorHist* (32-D) (b) *LayoutHist* (32-D)

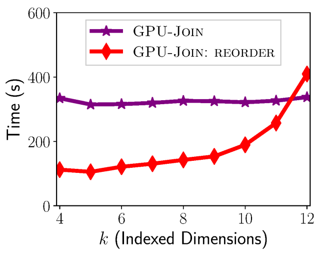
Figure 4: Response time vs. ϵ on the real-world datasets used in Super-EGO [33]. $k = 6$ dimensions are indexed. Values of S_D are in the range (a) 4–26k, and (b) 3–1.1k.

5.3.2 Index Dimension Reduction & Reordering. Recall from Section 4.1 that, by indexing $k < n$ dimensions, we reduce the search overhead but increase the number of necessary Euclidean distance calculations. We also reorder the points to exploit variance in the dimensions of the data (Section 4.2), maximizing the the filtering power of the dimensions that we do index.

Figure 5(a) plots the response time vs. k on *SuSy* where $\epsilon = 0.01$ with and without using reorder. In both cases, we see that



(a) *SuSy* (18-D), $\epsilon = 0.01$



(b) *Songs* (90-D), $\epsilon = 0.005$

Figure 5: Response time vs. indexed dimension, k.

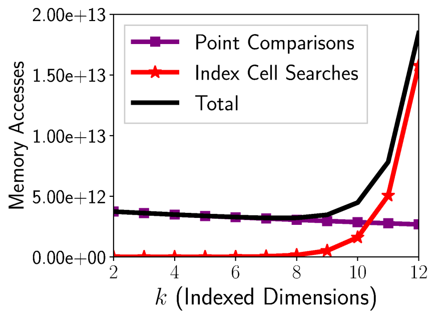


Figure 6: The number of index search and point comparison memory accesses on *Syn16D2M*.

performance degrades when too few or too many dimensions are indexed due to increased point comparisons or search overhead, respectively. When we reorder the data by variance, the response time is significantly reduced, particularly for small k. Since the data is indexed in the first k dimensions, when we do not use REORDER it is possible that the indexed dimensions will have high variance *by chance*, providing good performance. However, if they do not, performance will significantly degrade. In the worst case, variance is so small that we must perform $O(|D|^2)$ distance comparisons. In such a case, using REORDER can significantly improve the ability of the index to prune the search for points within ϵ. Figure 5(b) shows the same plot for the *Songs* dataset where $\epsilon = 0.005$. This is an example where the first $k \lesssim 12$ dimensions have low variance, thereby generating a grid with few cells and low index filtering power. We note that when $k = 12$, using REORDER *increases* response time compared to not using the optimization. This is because, while REORDER exploits variance to improve the index filtering power, it also increases the number of cells and therefore the search overhead. For either *SuSy* or *Songs*, REORDER significantly reduces the response time when $3 \leq k \leq 8$, which is a large range from which to select k. From these experimental results, we can simply select $k = 6$ dimensions. However, in the next section, we show how (and why) a good value of k can be selected.

5.3.3 The Number of Indexed Dimensions. Indexing $k < n$ dimensions provides a trade-off between distance comparisons and search overhead. We describe a method to select a good value of k. Real-world high-D datasets do not allow us to use analytical methods to estimate the amount of work needed to perform the self-join. For example, we cannot analytically compute the average number of cells searched, the average number of point comparisons, and the average number of neighbors per point. Thus, we use a sampling technique to understand these data-dependent characteristics.

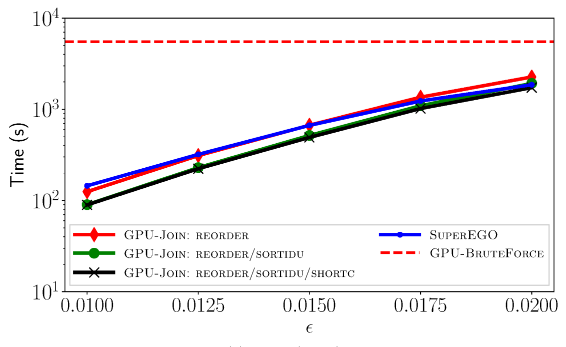
For a given value of k, we execute GPU-Join for a fraction f of the data points, and record the number of point comparisons (points that are tested to be within ϵ of each other), denoted as μ. With the sample, we estimate the total number of memory operations needed for the distance comparisons as $\mu \cdot \frac{1}{f}$. We can select a good value of k, by comparing the total number of memory operations for: (i) distance comparisons; and, (ii) searching whether the cells exist (i.e., $|D|3^k\log_2(|G|)$, where $|G|$ is the total number of non-empty cells).

Figure 6 plots the number of memory operations vs. k on *Syn16D2M*. A reduction in distance comparison memory operations occurs as k increases, indicating that indexing on more dimensions reduces the number of distance comparisons. Few memory operations are need for cell searches when $k \leq 8$, but the exponential increase in the number of adjacent cells with increasing k makes indexing $k > 10$ degrade performance. Regardless of dataset, indexing $k > 10$ is likely to degrade performance. Regarding index memory accesses only, in practice we can select a value of k within a fairly large range, $k \leq 10$, without significant performance loss. Thus, we do not need excessive parameter tuning of k to achieve good performance. Consequently, we index on $k = 6$ dimensions.

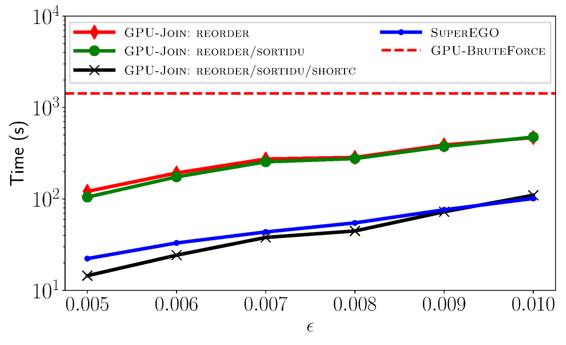
5.3.4 Larger Real World Datasets. Figure 7(a) plots the response time vs. ϵ on *SuSy*. Note that *SuSy* is two orders of magnitude larger ($|D| = 5 \times 10^6$) than those used in Figure 4. Results indicate that, for the *SuSy* dataset, SORTIDU reduces response time by a reasonable margin (e.g., at $\epsilon = 0.01$, using SORTIDU and REORDER is 38% faster than REORDER alone), though the SHORTC optimization has a negligible effect. Using all optimizations, GPU-Join outperforms Super-EGO across all values of ϵ, with speedups up to 1.61× at $\epsilon = 0.01$.

Figure 7(b) plots the response time vs. ϵ on the 90-D *Songs* dataset. We observe that using SORTIDU reduces the response time at lower values of ϵ. In contrast to the *SuSy* dataset (Figure 7(a)), SHORTC yields a significant reduction in response time on the *Songs* dataset (Figure 7(b)) due to its much higher dimensionality. We note that Super-EGO also employs an optimization that short circuits the distance calculation. The speedup (or slowdown) over Super-EGO ranges from 1.53× ($\epsilon = 0.005$) to 0.92× ($\epsilon = 0.01$). GPU-Join outperforms Super-EGO across all experiments, except for a slight slowdown on *Songs* at $\epsilon = 0.01$, indicating that Super-EGO is competitive with GPU-Join under some experimental scenarios. On the *SuSy* and *Songs* datasets, GPU-BruteForce does not outperform GPU-Join or Super-EGO.

5.3.5 Synthetic Datasets. We use synthetic datasets to understand when GPU-Join cannot exploit REORDER (Section 4.2) because the variance is nearly the same in each dimension. We utilize synthetic datasets with an exponential distribution as it ensures that we will find a reasonable number of neighbors for a given ϵ, similarly to the high-D real-world datasets. Figure 8 plots the response time vs. ϵ on

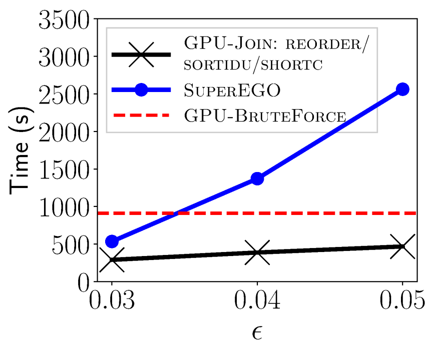


(a) *SuSy* (18-D)

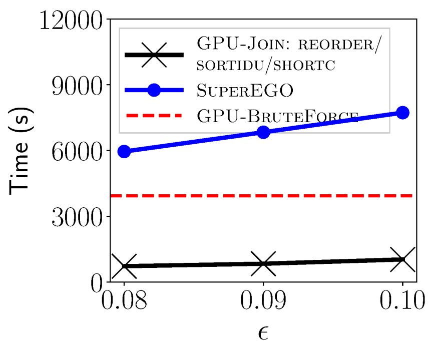


(b) *Songs* (90-D)

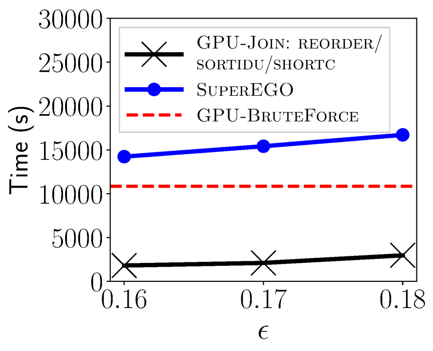
Figure 7: Response time vs. ϵ. $k = 6$ dimensions are indexed. Values of S_D are in the range (a) 5–781, and (b) 4–1.9k.



(a) *Syn16D2M* (16-D)



(b) *Syn32D2M* (32-D)



(c) *Syn64D2M* (64-D)

Figure 8: Response time vs. ϵ on synthetic datasets. $k = 6$ dimensions are indexed, and GPU-Join is configured using all optimizations. Values of S_D are in the range (a) 4–1.2k, (b) 31–1.4k, and (c) 132–2.3k.

16, 32, and 64-D synthetic datasets. GPU-Join outperforms Super-EGO on all scenarios, with the smallest performance gain a speedup of 1.84× on the smallest workload (Figure 8(a), $\epsilon = 0.03$) and the largest speedup of 8.25× on *Syn32D2M* with $\epsilon = 0.08$. Both Super-EGO and GPU-Join exploit the statistical properties of the data. However, these results indicate that the index search performance of Super-EGO is more dependent on the statistical properties of the data. Thus, in cases where the variance is similar across dimensions, GPU-Join is likely to significantly outperform Super-EGO. We also find that GPU-Join outperforms GPU-BruteForce across all datasets in Figure 8, which implies that GPU-Join index search performance does not degrade to a brute force search on these datasets.

6 DISCUSSION & CONCLUSIONS

In this work, we propose GPU-Join, an algorithm that leverages the GPU's massive parallelism and high memory bandwidth to efficiently solve the self-join problem. We show that a grid-based index, combined with index dimensionality reduction (indexing $k < n$), reordering the data by the variance in each dimension (reorder), and distance calculation reduction (sortidu and shortc) significantly improves self-join performance over Super-EGO in most experimental scenarios.

In Section 5, we demonstrated that GPU-Join is less sensitive to the data distribution than the state-of-the-art Super-EGO. The characteristics of the dataset set nevertheless have a significant impact on the performance of GPU-Join. This is because the efficacy of the indexing structures to prune the search in both Super-EGO and GPU-Join depend on the variance of the data, which is a function of the underlying data distribution.

In general, there are several common GPU bottlenecks that may impact the performance of GPU-Join that depend on the data distribution. The SIMT execution of modern GPUs requires that groups of threads execute the same instructions to achieve peak performance [40]. Thus, the data distribution has a direct impact on the amount of warp divergence that occurs in GPU-Join. Recently, for low-dimensional distance similarity self-joins, it was shown that threads with varying numbers of candidate points (or workloads) will cause intra-warp load imbalance and divergence [25]. While this work was for the low-dimensional case [25], intra-warp load imbalance will have an impact on GPU-Join performance for the high-dimensional case as well. Additionally, many of the optimizations discussed in Section 4 depend on statistical properties of the dataset to improve performance (e.g., reorder). Consequently, an interesting direction of future work is to develop a model of the workload and divergence as a function of the input data distribution. Such a model would provide insight into the performance bottlenecks of GPU-Join and other indexing structures on the GPU. These insights can help us develop more optimizations that further improve GPU-Join performance and alternate approaches of solving the self-join problem on datasets for which existing methods (e.g., Super-EGO or GPU-Join) may not perform well.

ACKNOWLEDGMENTS

We thank University of Hawaii HPC for platform use. This material is based upon work supported by the National Science Foundation under Grants 1849559, 1533823, and 1745331 and Fonds de la Recherche Scientifique-FNRS under Grant no MISU F 6001 1.

REFERENCES

[1] [n.d.]. Nvidia Volta. http://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf. Accessed: Oct. 5, 2018.

[2] [n.d.]. Top500. https://www.top500.org/lists/2018/06/. Accessed: Apr. 29, 2019.

[3] Rakesh Agrawal, Christos Faloutsos, and Arun Swami. 1993. Efficient similarity search in sequence databases. *Foundations of data organization and algorithms* (1993), 69–84.

[4] Alexandr Andoni and Piotr Indyk. 2006. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. In *IEEE Symp. on Foundations of Computer Science*. 459–468.

[5] Mihael Ankerst, Markus M. Breunig, Hans-Peter Kriegel, and Jörg Sander. 1999. OPTICS: Ordering Points to Identify the Clustering Structure. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*. 49–60.

[6] Arvind Arasu, Venkatesh Ganti, and Raghav Kaushik. 2006. Efficient Exact Set-similarity Joins. In *Proc. of the Intl. Conf. on Very Large Data Bases*. 918–929.

[7] P. Baldi, P. Sadowski, and D. Whiteson. 2014. Searching for exotic particles in high-energy physics with deep learning. *Nature Communications* 5, Article 4308 (July 2014). arXiv:hep-ph/1402.4735

[8] Roberto J. Bayardo, Yiming Ma, and Ramakrishnan Srikant. 2007. Scaling Up All Pairs Similarity Search. In *Proc. of the Intl. Conf. on World Wide Web*. 131–140.

[9] Richard E Bellman. 1961. *Adaptive control processes: a guided tour.* Princeton University press.

[10] Jon Louis Bentley. 1975. Multidimensional Binary Search Trees Used for Associative Searching. *Commun. ACM* 18, 9 (1975), 509–517.

[11] Thierry Bertin-Mahieux, Daniel P.W. Ellis, Brian Whitman, and Paul Lamere. 2011. The Million Song Dataset. In *Proc. of the 12th Intl. Conf. on Music Information Retrieval.*

[12] Christian Böhm, Bernhard Braunmüller, Markus Breunig, and Hans-Peter Kriegel. 2000. High Performance Clustering Based on the Similarity Join. In *Proc. of the Intl. Conf. on Information and Knowledge Management*. 298–305.

[13] Christian Böhm, Bernhard Braunmüller, Florian Krebs, and Hans-Peter Kriegel. 2001. Epsilon grid order: An algorithm for the similarity join on massive high-dimensional data. In *ACM SIGMOD Record*, Vol. 30. 379–388.

[14] C. Bohm and H. P. Kriegel. 2001. A cost model and index architecture for the similarity join. In *Proc. 17th Intl. Conf. on Data Engineering*. 411–420.

[15] Christian Böhm, Robert Noll, Claudia Plant, and Andrew Zherdin. 2009. Index-supported Similarity Join on Graphics Processors. In *BTW*. 57–66.

[16] Brent Bryan, Frederick Eberhardt, and Christos Faloutsos. 2008. Compact similarity joins. In *IEEE 24th Intl. Conf. on Data Engineering*. 346–355.

[17] Gang Chen, Keyu Yang, Lu Chen, Yunjun Gao, Baihua Zheng, and Chun Chen. 2017. Metric similarity joins using MapReduce. *IEEE Transactions on Knowledge and Data Engineering* 29, 3 (2017), 656–669.

[18] Akash Das Sarma, Yeye He, and Surajit Chaudhuri. 2014. ClusterJoin: A Similarity Joins Framework using Map-Reduce. *Proceedings of the VLDB Endowment* 7, 12 (2014), 1059–1070.

[19] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. *CACM* 51, 1 (2008), 107–113.

[20] Jens-Peter Dittrich and Bernhard Seeger. 2001. GESS: A Scalable Similarity-join Algorithm for Mining Large Data Sets in High Dimensional Spaces. In *Proc. of the ACM Intl. Conf. on Knowledge Discovery and Data Mining*. 47–56.

[21] Robert J. Durrant and Ata Kabán. 2009. When is 'Nearest Neighbour' Meaningful: A Converse Theorem and Implications. *Journal of Complexity* 25, 4 (2009), 385–397.

[22] Martin Ester, Hans Kriegel, Jörg Sander, and Xiaowei Xu. 1996. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Proc. of the 2nd KDD*. 226–231.

[23] Raphael A. Finkel and Jon Louis Bentley. 1974. Quad trees a data structure for retrieval on composite keys. *Acta informatica* 4, 1 (1974), 1–9.

[24] S. Fries, B. Boden, G. Stepien, and T. Seidl. 2014. PHiDJ: Parallel similarity self-join for high-dimensional vector data with MapReduce. In *2014 IEEE 30th Intl. Conf. on Data Engineering*. 796–807.

[25] Benoit Gallet and Michael Gowanlock. 2019. Load Imbalance Mitigation Optimizations for GPU-Accelerated Similarity Joins. In *Proc. of the 2019 IEEE Intl. Parallel and Distributed Processing Symp. Workshops (IPDPSW), to appear.*

[26] Michael Gowanlock and Henri Casanova. 2016. Distance Threshold Similarity Searches: Efficient Trajectory Indexing on the GPU. *IEEE Transactions on Parallel and Distributed Systems* 27, 9 (2016), 2533–2545.

[27] M. Gowanlock and B. Karsin. 2018. GPU Accelerated Self-Join for the Distance Similarity Metric. In *2018 IEEE Intl. Parallel and Distributed Processing Symp. Workshops (IPDPSW)*. 477–486.

[28] Michael Gowanlock, Cody M Rude, David M Blair, Justin D Li, and Victor Pankratius. 2017. Clustering Throughput Optimization on the GPU. In *Proc. of the IEEE Intl. Parallel and Distributed Processing Symp.* 832–841.

[29] Antonin Guttman. 1984. R-trees: a dynamic index structure for spatial searching. In *Proc. of Intl. Conf. on Management of Data*. 47–57.

[30] Gibran Hemani, Athanasios Theocharidis, Wenhua Wei, and Chris Haley. 2011. EpiGPU: exhaustive pairwise epistasis scans parallelized on consumer level graphics cards. *Bioinformatics* 27, 11 (2011), 1462–1465.

[31] Edwin H. Jacox and Hanan Samet. 2007. Spatial Join Techniques. *ACM Trans. Database Syst.* 32, 1, Article 7 (2007).

[32] Edwin H. Jacox and Hanan Samet. 2008. Metric Space Similarity Joins. *ACM Trans. Database Syst.* 33, 2, Article 7 (2008), 38 pages.

[33] Dmitri V Kalashnikov. 2013. Super-EGO: fast multi-dimensional similarity join. *The VLDB Journal* 22, 4 (2013), 561–585.

[34] Jinwoong Kim, Won-Ki Jeong, and Beomseok Nam. 2015. Exploiting Massive Parallelism for Indexing Multi-Dimensional Datasets on the GPU. *IEEE Transactions on Parallel and Distributed Systems* 26, 8 (2015), 2258–2271.

[35] Jinwoong Kim and Beomseok Nam. 2018. Co-processing heterogeneous parallel index for multi-dimensional datasets. *J. Parallel and Distrib. Comput.* 113 (2018), 195–203.

[36] Krzysztof Koperski and Jiawei Han. 1995. Discovery of spatial association rules in geographic information databases. In *Advances in spatial databases*. Springer, 47–66.

[37] Francesco Lettich, Salvatore Orlando, Claudio Silvestri, and Christian S Jensen. 2017. Manycore GPU processing of repeated range queries over streams of moving objects observations. *Concurrency and Computation: Practice and Experience* 29, 4 (2017), e3881.

[38] M. Lichman. 2013. UCI Machine Learning Repository. http://archive.ics.uci.edu/ml

[39] Michael D Lieberman, Jagan Sankaranarayanan, and Hanan Samet. 2008. A fast similarity join algorithm using graphics processing units. In *IEEE 24th Intl. Conf. on Data Engineering*. 1111–1120.

[40] NVIDIA. 2018. Pascal Tuning Guide. http://docs.nvidia.com/cuda/pascal-tuning-guide/index.html. Accessed 18-June-2018.

[41] S. K. Prasad, M. McDermott, X. He, and S. Puri. 2015. GPU-based Parallel R-tree Construction and Querying. In *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*. 618–627.

[42] Mahsan Rofouei, Thanos Stathopoulos, Sebi Ryffel, William Kaiser, and Majid Sarrafzadeh. 2008. Energy-aware high performance computing with graphic processing units. In *Workshop on power aware computing and system.*

[43] Thomas Seidl, Sergej Fries, and Brigitte Boden. 2013. MR-DSJ: Distance-Based Self-Join for Large-Scale Vector Data Analysis with MapReduce. In *BTW*, Vol. 214. 37–56.

[44] H. So, J. Chen, B. Yiu, and A. Yu. 2011. Medical Ultrasound Imaging: To GPU or Not to GPU? *IEEE Micro* 31, 5 (2011), 54–65.

[45] Ilya Volnyansky and Vladimir Pestov. 2009. Curse of Dimensionality in Pivot Based Indexes. In *Proc. of the Second Intl. Workshop on Similarity Search and Applications*. 39–46.

[46] Jianting Zhang, Simin You, and Le Gruenwald. 2012. U^{2}STRA: High-performance Data Management of Ubiquitous Urban Sensing Trajectories on GPGPUs. In *Proc. of the ACM Workshop on City Data Management*. 5–12.