A Coordinate-Oblivious Index for High-Dimensional Distance Similarity Searches on the GPU

Brian Donnelly
Brian.Donnelly@nau.edu
Northern Arizona University
Flagstaff, Arizona, USA

Michael Gowanlock
Michael.Gowanlock@nau.edu
Northern Arizona University
Flagstaff, Arizona, USA

ABSTRACT

We present COSS, an exact method for high-dimensional distance similarity self-joins using the GPU, which finds all points within a search distance ϵ from each point in a dataset. The similarity self-join can take advantage of the massive parallelism afforded by GPUs, as each point can be searched in parallel. Despite high GPU throughput, distance similarity self-joins exhibit irregular memory access patterns which yield branch divergence and other performance limiting factors. Consequently, we propose several GPU optimizations to improve self-join query throughput, including an index designed for GPU architecture. As data dimensionality increases, the search space increases exponentially. Therefore, to find a reasonable number of neighbors for each point in the dataset, ϵ may need to be large. The majority of indexing strategies that are used to prune the ϵ-search focus on a spatial partition of data points based on each point's coordinates. As dimensionality increases, this data partitioning and pruning strategy yields exhaustive searches that eventually degrade to a brute force (quadratic) search, which is the well-known curse of dimensionality problem. To enable pruning the search using an indexing scheme in high-dimensional spaces, we depart from previous indexing approaches, and propose an indexing strategy that does not index based on each point's coordinate values. Instead, we index based on the distances to reference points, which are arbitrary points in the coordinate space. We show that our indexing scheme is able to prune the search for nearby points in high-dimensional spaces where other approaches yield high performance degradation. COSS achieves a speedup over CPU and GPU reference implementations up to 17.7× and 11.8×, respectively.

CCS CONCEPTS

• **Computing methodologies → Massively parallel algorithms**;
• **Information systems → Data mining**.

KEYWORDS

GPU, High Dimensional, In-memory Database, Multidimensional Index, Similarity Search

ACM Reference Format:
Brian Donnelly and Michael Gowanlock. 2020. A Coordinate-Oblivious Index for High-Dimensional Distance Similarity Searches on the GPU. In *2020 International Conference on Supercomputing (ICS '20), June 29-July 2, 2020, Barcelona, Spain.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3392717.3392768

1 INTRODUCTION

Similarity searches are fundamental database operations and are used in data analysis. For example, similarity searches [22, 29, 30, 32] are used in clustering algorithms [31], and k-nearest-neighbors searches [1, 8]. This paper examines the distance similarity self-join problem [10, 11, 22, 27], defined as searching a distance ϵ around each point in a dataset and returning all of the neighbors within this search distance. We focus on a GPU-efficient, coordinate-oblivious index that prunes the search for nearby points. While we use the index for the distance similarity self-join, the index can be employed in other spatial search algorithms.

A semi-join on two datasets $A \ltimes_{\epsilon} B$ involves comparing every point in A to every point in B with a complexity $O(|A| \cdot |B|)$. Comparatively, self-joins ($A \bowtie_{\epsilon} A$) involve comparing all of the points in a single dataset with a complexity $O(|A|^2)$. In this paper, we examine the self-join, but note that the method and most optimizations proposed can be employed for the semi-join as well.

The brute force approach to the distance similarity self-join computes the distance from every point to every other point yielding a time complexity of $O(n^2)$, where n is the number of data points in a dataset, making the approach impractical for large datasets. Index-trees use the data's coordinate values to build a hierarchical data structure of partitions. For example, kd-trees [13, 36], R-trees (and R-tree variants) [3, 9, 14–17, 20, 23, 24, 28, 35], and X-trees [5] are all types of trees that prune the search for nearby objects and are optimized for specific application scenarios. Grid-based indexes with fixed length cells [11, 12, 19, 27, 29] have also been proposed to partition the dataset. The major difference between index-trees and grids is that many index-trees construct the index based on the positions of the points, whereas static grids partition the space independently of the data distribution.

Both trees [20, 28] and grids [11, 12, 19, 27, 29] have been designed for the GPU. Searching an index on the GPU introduces several challenges related to both index types. For example, searches on trees require tree traversals which may lead to divergent execution paths that degrade performance on GPUs [20, 26]. Depending on the type of query, a static grid may perform worse than a tree, because the data partitions are of equal size. For the self-join problem with a fixed search radius, static grids are an attractive option because ϵ-length cell sizes can be utilized, which bound the search

ICS '20, June 29-July 2, 2020, Barcelona, Spain

ACM ISBN 978-1-4503-7983-0/20/06…$15.00
https://doi.org/10.1145/3392717.3392768

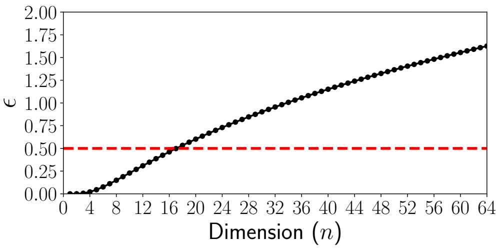


Figure 1: ϵ vs. n showing the minimum value of ϵ required to find a single neighbor (on average) within a unit n-dimensional hypercube, $\beta = [0, 1]^n$, where the $|D| = 10^6$ data points are uniformly distributed. When $\epsilon \geq 0.5$ (red dashed line), a grid-based index degrades to a brute-force search.

to neighboring cells [19]. Additionally, grids may have less branch divergence than trees, since trees require many branch conditions in their traversals [20, 28].

The volume of the space that needs to be searched grows exponentially with data dimensionality. To find points near each other, the search distance ϵ needs to increase proportionately to the increase in dimensionality. Figure 1 shows the ϵ needed to find one average neighbor on a uniformly distributed dataset. We observe that as the dimensionality of the data increases, ϵ has to increase to maintain finding a single neighbor. In a grid-based index the search within a unit hypercube becomes brute force when $\epsilon = 0.5$, which occurs at only 18 dimensions (Section 3.1). This illustration shows that the pruning efficacy of methods that index based on the coordinate space of the data (e.g., grids and trees) causes most index searches to degrade rapidly into a brute force search. This is known as the *curse of dimensionality* problem [4].

The ability to efficiently use the GPU makes grid-indexing a good solution for large dataset analysis. With a higher memory bandwidth, and a massive throughput for floating point calculations [25], GPUs provide the ability to replace large multi-core systems with a single device [34].

We propose COSS– a GPU algorithm for high-dimensional similarity searches. GPUs have high memory bandwidth and high throughput for floating point calculations [25], which are needed to compute Euclidean distances between neighbors. COSS is designed to address high-dimensional similarity searches by constructing a coordinate-oblivious index in distance space. COSS indexes based on the distance to an arbitrary point in space, that we denote as a reference point. Data points are then ordered and assigned to bins based on this distance. COSS has a similar instruction flow as searches on grids, but does not partition on coordinate space. By indexing based on the distance to a reference point, we can construct an index that does not rely on the individual coordinates of the data, but instead utilizes the entire set of coordinate values for indexing. This reduces the curse of dimensionality problem described above. We show that COSS is a more efficient algorithm than other state-of-the-art methods for high-dimensional distance similarity self-joins. We outline the major contributions of this paper as follows:

- We propose a novel coordinate-oblivious indexing method, COSS, tailored to exact similarity searches on high dimensional data using the GPU.
- By using our coordinate-oblivious indexing scheme, we optimize pruning power by selecting the number of reference points and their location.
- We leverage several optimization that improve index performance and memory management, including dimensional ordering, short circuiting the distance calculations, and batching the computation across multiple kernel invocations.
- We evaluate COSS on 3 real-world datasets, 2 synthetic datasets and compare to other state-of-the-art methods, Super-EGO and GPU-Join.

The paper is organized as follows. Section 2 presents the problem statement, Section 3 discusses the curse of dimensionality problem and related work, Section 4 presents our coordinate-oblivious indexing scheme, Section 5 presents the optimizations used in COSS, Section 6 presents our results, and finally, Section 7 concludes the paper.

2 PROBLEM STATEMENT

We outline the distance similarity self-join problem, denoted as $D \bowtie_\epsilon D$, as follows. Let D be a dataset, containing $|D|$ points (or feature vectors), in n dimensions. Each point is defined as $p_i \in D$, where $i = 1, 2, \ldots, |D|$. We denote the coordinates of each point, $p_i \in D$, as $p_i = (x_1, x_2, \ldots, x_n)$. Like other works [5, 11, 12, 17, 19, 27, 29] we use the Euclidean distance similarity measure. The Euclidean distance between points $r \in D$ and $s \in D$ is defined as $dist(r, s) = \sqrt{\sum_{j=1}^{n}(r_j - s_j)^2}$. The self-join performs similarity searches on all points in the dataset, $p_i \in D$. A pair of points r and s are added to the result set if $dist(r, s) \leq \epsilon$. The value of ϵ directly controls the selectivity of the self-join, where the selectivity refers to the average number of neighbors found per point in the dataset, D.

In this paper, all processing occurs in-memory. We consider the case where the result set size may exceed the GPU's global memory capacity instead of limiting our work to the case where the result set must fit within global memory on the device. Since the result set size is typically much larger than the input dataset size, we do not allow for the case where the input dataset exceeds global memory capacity.

3 BACKGROUND

In this section, we provide an overview of the motivation and literature. We use CUDA terminology throughout this section and paper.

3.1 Motivation: Selectivity and the Curse of Dimensionality

We illustrate the relationship between dimensionality (n), ϵ, and selectivity, where selectivity refers to the average number of neighbors found by each point. We denote selectivity as $S = (|R| - |D|)/|D|$ where R is the result set and D is the input dataset. We draw on the example given by Kalashnikov [19], and refer the reader to that paper for a comprehensive discussion of selectivity. Consider a unit hypercube containing $|D| = 10^6$ uniformly distributed data points in the bounding volume defined by $\beta = [0,1]^n$. Because points are uniformly distributed in the hypercube, as the dimensionality, n, increases, the search distance ϵ will need to increase to find neighboring points. Assume that we wish to find 1 neighbor on average (i.e., a selectivity $S = 1$).

We compute the value of ϵ needed to find $S = 1$ using geometric arguments. First, we define the volume of an n-dimensional sphere with radius ϵ as follows: $V(n, \epsilon) = g(n)\epsilon^n$, where $g(n) = \frac{\pi^{n/2}}{\Gamma(\frac{n}{2}+1)}$. If the volume needed to search a query point lies entirely within β, and is not positioned near the edge of the bounding volume, β, then the point is more likely to find neighbors within its search radius. We consider this best case scenario for a given search.

To find a selectivity of $S = 1$ point on average, we want to find the value of ϵ where $|D| \cdot g(n)\epsilon^n \geq 1$. Solving for ϵ, we obtain $\epsilon \geq (|D| \cdot g(n))^{-1/n}$. Figure 1 plots ϵ vs. dimension (n), where a value of ϵ below that, plotted for a given value of n, yields $S < 1$ (less than 1 neighbor found per point on average). Since grid indexing schemes constrain the search to adjacent grid cells, then a search in β with $\epsilon \geq 0.5$ will degrade to a brute force search because searching adjacent grid cells will span the entire bounding volume, β [19]. From the plot, we find that at $n = 18$, $\epsilon \geq 0.53$ is needed to find a single neighbor. Consequently, for uniformly distributed data, indexing the data based on their coordinate values will degrade to a brute force search when $n \geq 18$ dimensions. This illustrative example shows the pitfalls of using grid-indexing schemes for high-dimensional data. Additionally, other methods that index the data based on a point's coordinate values, such as index-trees (e.g., R-tree [14], X-tree [5], kd-tree [36]), suffer from the same curse of dimensionality problem.

3.1.1 *Dimensionality Reduction and Approximate Solutions.*

One method of processing high-dimensional datasets and counteracting the curse of dimensionality is to use a feature extraction method like Principle Component Analysis [33] or Map Analysis [8]. While reducing the effective dimensions of the data is a straightforward method for reducing the computation time, there is a loss of data that results in approximate solutions. For an exact solution, a larger amount of computation is needed [11], and we focus on exact similarity searches in this paper.

3.2 Related Work

Indexing methods can reduce the runtime of a distance similarity self-join by reducing the total number of distance calculations needed [17]. Indexing methods partition the input dataset, allowing the algorithms to prune the search space by only evaluating nearby searched query points. There are two main approaches to indexing: one is to construct an index using the coordinate values of the points (e.g., kd-trees [36]), and the other is a data oblivious approach that creates the index by partitioning the space (e.g., statically partitioned grids [19]). Our COSS algorithm is differentiated by its coordinate-oblivious index; COSS does not rely on either the data coordinate values or partitioning the coordinate space to construct the index.

3.2.1 *Index-trees.*

Trees construct a hierarchical index that partitions the coordinate space. [3, 5, 9, 13–17, 20, 23, 24, 28, 35, 36]. For example, in an R-tree, when a query point is being searched, the tree is traversed to find points within the query point's minimum bounding box. When concurrently searching the tree, traversals cause thread divergence on the GPU because of irregular instruction flow [20, 28]. Several methods have been developed to improve tree searching performance on the GPU. For example, Kim et al. [20] propose a technique that allows trees to search on the GPU, minimizing the divergence and avoiding back-tracking.

While most CPU index-trees use a depth first search, GPU implementations use a breadth first search to help reduce branching [28]. The downside of the breadth first search is that it can require a large amount of dynamic storage. [28] Large storage requirements are problematic because of the limited amount of memory available on the GPU. Even with a number of optimizations made for index-trees that use the GPU, the architecture of the GPU may not be well-suited to index tree searches.

3.2.2 *Grid-based Indexes.*

Grid-based indexing methods [12, 19, 27] build a structure that partitions the space and then assigns points to a cell based on their coordinate values. The index itself is constructed in a data oblivious manner, but the points are assigned to the cells based on the coordinate values of the points. In contrast to the R-tree, grid-based index searches use a deterministic instruction flow when checking adjacent cells. This makes grid-based index searching more well-suited to the GPU architecture. ϵ Grid Order [7] indexes use cells that have edges of length ϵ, when ϵ becomes a large portion of the range in a single dimension, the number of total cells decreases along with pruning efficiency. We discuss two grid-based implementations in the following section.

3.2.3 *The iDistance Method.*

Jagedish et al. [18] propose the iDistance method which creates an adaptive B$^+$-tree by indexing the points on distance to a reference point in the coordinate space. Each point in the dataset is assigned to the closest reference point. A one-dimensional B$^+$-tree is constructed using the distance from each data point to its assigned reference point. This indexes the data on a single dimension based on distance to the nearest reference point. In contrast to indexing directly on the coordinate space of the data, points that are adjacent in the iDistance B$^+$-tree may not be adjacent in the coordinate space.

Similarly to COSS, the iDistance method uses the distance to reference points to construct an index. In contrast to our proposed algorithm, iDistance creates a one-dimensional tree using multiple reference points, while COSS creates a multi-dimensional grid-like index. The data points are assigned locations in the index based on their distance to every reference point in the COSS algorithm, while iDistance only uses the distance to a single reference point for each data point. Therefore, COSS has the ability to increase pruning capability compared to iDistance. While iDistance is not implemented on the GPU, the B$^+$-tree structure would have the same problems as other tree-indexes as discussed above, while COSS is designed specifically to exploit GPU hardware.

3.3 Reference Implementations

We compare COSS to two state-of-the-art reference implementations Super-EGO and GPU-Join. We review the two methods below.

3.3.1 *Super-EGO.*

Kalashnikov's Super-EGO [19] is a CPU-only grid-indexing method that indexes dimensions intelligently. By carefully selecting which dimension to index, the Super-EGO algorithm is able to increase pruning. One weakness of the algorithm pointed out by the authors is that, for datsets normalized to $[0, 1]^n$, any $\epsilon \geq 0.5$ causes the runtime to become quadratic. This is a similar weakness shared by other grid-based indexes and index-trees which our method addresses. In this paper we use Super-EGO as a reference for evaluating the performance of COSS.

3.3.2 GPU-Join. Gowanlock and Karsin [12] introduce a GPU grid-index for joins that has several optimizations to improve performance on high-dimensional datasets. GPU-Join reduces the number of indexed dimensions to avoid increasing the cost of index searches, while this increases the number of distance calculations, it reduces the overall work. When reducing the amount of partitioning, the algorithm uses statistics to decide which subset of the dimensions to index on, thereby maximizing the pruning effectiveness of the index. These optimizations allow GPU-Join to address high-dimensional datasets.

4 INDEXING ON DISTANCE SPACES FOR HIGH-DIMENSIONAL DATA

4.1 Overview: Indexing by Distance to Points

To mitigate the curse of dimensionality (see Section 3.1), we can construct a coordinate-oblivious index. Our proposed index uses the distance to an arbitrary point in the coordinate space. We call this arbitrary point a reference point and find the distance between it and every other point in the dataset. Figure 2 shows the *distance space* with a reference point *RP* and 10 data points. The distance from the reference point is segmented into ϵ-width bins. The point p_4 in Figure 2 is in the second bin, so we know that there is no possibility of it being within ϵ of p_8 which is in the fourth bin. Searches for points within ϵ of p_4 can prune any points that are not in bins 1, 2 or 3, because points in other bins exceed the search distance ϵ.

We refer to the *distance space* as the location of each $p_i \in D$ based on its distance to the reference points (e.g., Figure 2 is a 1-D distance space, and Figure 3 shows a 2-D distance space). We refer to the *coordinate space* as the typical Cartesian space that contains the input dataset point coordinates of each $p_i \in D$.

The index stores the bins that each point is located in. The number line in Figure 2 shows how the points would be placed into bins based on their distance to the reference point (*RP*). In Section 4.2.1 we show how we use the distance to a reference point to construct the index.

By indexing with the distance to a reference point, we avoid relying on the coordinate space to partition the data. This method directly addresses the issue that arises from selectivity and the curse of dimensionality discussed in Section 3.1. This index is still affected by the increase in the dimensionality of the data, but only insomuch as that it affects the distances between points. The distance space is entirely independent of the dimensionality of the data. Consequently, this yields an opportunity to have a higher pruning capacity than methods that index on the coordinate space (e.g., index-trees and grids).

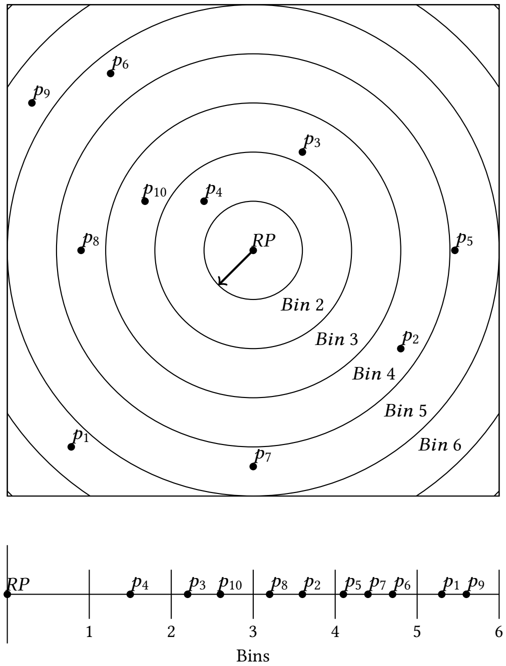


Figure 2: This figure shows a graphical example of an index with a single reference point, *RP*. The number line in the bottom of the image shows where points fall into ϵ-width bins based on their distance to *RP*.

4.2 Bin and Index Construction

4.2.1 Reference Point Bin Construction. We create a set of W reference points, where the reference points are denoted as l_t, where $W = (l_1, l_2, \ldots, l_{|W|})$ and has the coordinates $l_t = (x_1, x_2, \ldots, x_n)$. We construct all reference point bins, $B = (B_1, B_2, \ldots, B_{|W|})$, by computing the the Euclidean distance between all $p_i \in D$ to all reference points, W. We define an array Q, where $|Q| = |D|$, which contains the point ids. We then stable sort the arrays B (keys) and Q (values) as key-value pairs. This is repeated $|W|$ times, each time using a different subset of B. Consequently, points that are within the same bin are stored contiguously in Q.

Example Bin Construction: In Figure 3 we construct an array B for the bins using two reference points RP_1, and RP_2, for an example dataset $D = (p_1, p_2, \ldots, p_{10})$, where $|D| = 10$. In step 1, we start with RP_2, and find the distance from every point $p_i \in D$ to RP_2, finding which bin each point falls into. We store the bin number for each point in array B_2, and store the corresponding point ids in Q. After B_2 has been computed, we sort Q and B_2 with a stable key-value sort that uses the bin numbers in B_2 as the key. This gives us an ordered array Q that starts with points in the lowest bin number and ends with points in the highest bin number. Arrays B_2 and Q in step 1 of Figure 3 show this sorted state.

In step 2 we consider RP_1 and repeat the procedure in step 1. When we use the stable sort on B_1 and Q, the points will maintain the order from B_2 in step 1 within the individual bins of B_1. This gives a final array B that is sorted from lowest to highest bin. Note that while p_2, p_6 and p_9 are spatially far apart, we can see that they have the same final bin numbers in the B array. This illustrates that points may be within the same bin in our index (nearby in the distance space) but distant in the coordinate space.

After B has been constructed, B contains duplicate bin ids. We reduce B to remove the duplicate bin ids to create B', such that we do not store redundant bin ids in the array; therefore, B' contains all of the non-empty and unique bin ids. To keep track of which points are in each bin, we construct a range array C as will be discussed in Section 4.2.2.

4.2.2 Index Construction. We compute the Euclidean distance between each reference point in W and $p_i \in D$, yielding the bin that contains each point. Using this information, we sort the points based on bin and store this information in Q. Next, we store B' (constructed as described in Section 4.2.1) which contains the unique bin ids (since many points may fall within a single bin, and some bins are empty, we only store the ids of the non-empty bins). We construct an array A that maps Q to B', which indicates the bin id of each point id in Q. For example, the point in $Q[i]$ is stored in the bin at $B'[A[i]]$. We construct an array C where $|C| = |B'|$ and $C[A[i]]$ contains the range of points in Q that are stored in bin $B'[A[i]]$. We illustrate the components of the index, when we show an example search in the next section.

4.3 Searching the Index

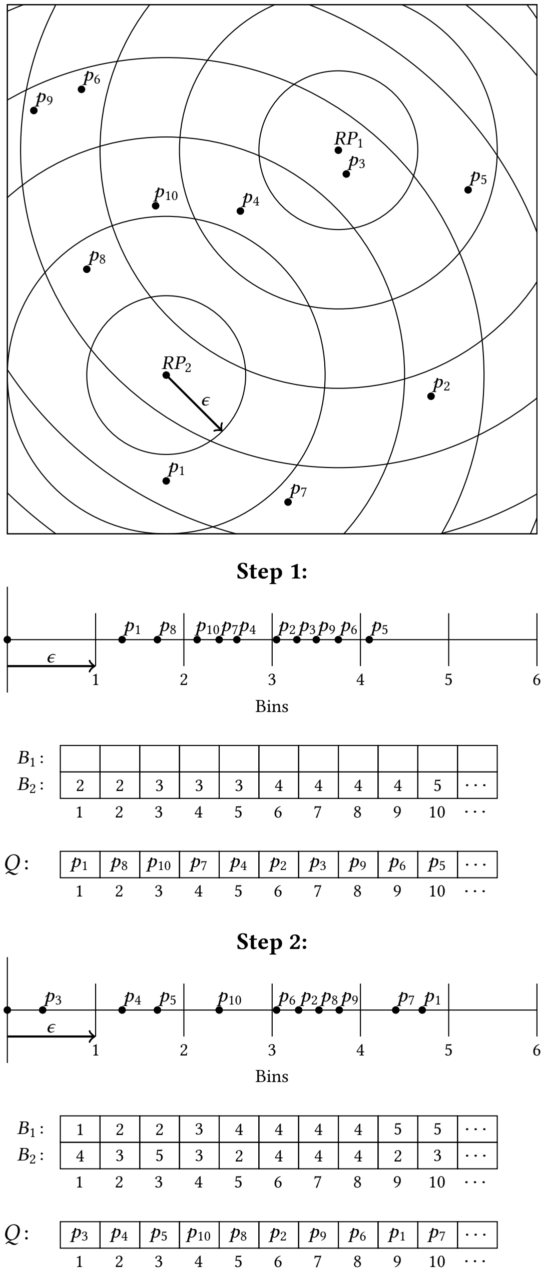
Each $p_i \in D$ is located within a single bin, where each bin is defined by $|W|$ bin numbers. Each bin has an address corresponding to the bin numbers. For example, a bin constructed with two reference points has bin numbers y_1 and y_2; therefore, adjacent bins are in the ranges $[y_1 - 1, y_1 + 1]$ and $[y_2 - 1, y_2 + 1]$. All the points in a bin will only need to evaluate the distance to points in the same, or adjacent bins as non-adjacent bins are separated by a distance $\geq \epsilon$.

We refer to a *query point* as a point in the dataset that is being searched. To find the adjacent bins for a query point, we take the query point's bin and compute the adjacent bin numbers (described above). We then do a binary search on array B' for those bin numbers. Note that B' only contains non-empty bins, so only a fraction of searches find a non-empty bin. Increasing the number of reference points increases the number of binary searches, as each query point executes $3^{|W|}$ binary searches.

Since we compute the self-join, we can eliminate duplicate distance calculations using the reflexive property (i.e., $dist(r, s) = dist(s, r)$), which reduces the total work by roughly half. To eliminate these distance calculations, we use the Unidirectional Comparison strategy developed by Gowanlock and Karsin [11] to select which bin numbers each query point will need to search. In short, the method halves the average number of adjacent bin searches. We refer the reader to Gowanlock and Karsin [11] for more detail. This optimization does not apply to the semi-join problem, only the self-join. All other optimizations (described in Section 5) can be applied to both the self-join and semi-join problems.

Example Search: Immediately after index construction, we transform D into D' by key-value sorting based on Q. This causes all of the coordinate data in D' to be mapped to the indices of Q.

For clarity, we outline an example search of our index without the Unidirectional Comparison [11] strategy and only index using a single reference point. Figure 4 shows an example of a search to find those points within ϵ of the query point $p_{61} \in D$. Point p_{61} at $Q[25]$ maps to $B'[9]$ using mapping array A. Since p_{61} is found in bin 48, we need to search adjacent bins, yielding a bin range of $[47, 49]$ (the three arrows from A to B'). Bins 47, 48, and 49 are found in C_h = 8, 9, 10, respectively. Note that the non-empty bins are not stored



B_1:											
B_2:	2	2	3	3	3	4	4	4	4	5	···
	1	2	3	4	5	6	7	8	9	10	···

Q:	p_1	p_8	p_{10}	p_7	p_4	p_2	p_3	p_9	p_6	p_5	···
	1	2	3	4	5	6	7	8	9	10	···

B_1:	1	2	2	3	4	4	4	4	5	5	···
B_2:	4	3	5	3	2	4	4	4	2	3	···
	1	2	3	4	5	6	7	8	9	10	···

Q:	p_3	p_4	p_5	p_{10}	p_8	p_2	p_9	p_6	p_1	p_7	···
	1	2	3	4	5	6	7	8	9	10	···

Figure 3: Example showing the construction of B and Q for two reference points. Every additional reference point adds an additional construction step. Note that the distance to a reference point within a bin does not impact the sorting, but the stable sort maintains the ordering from previous steps within a bin.

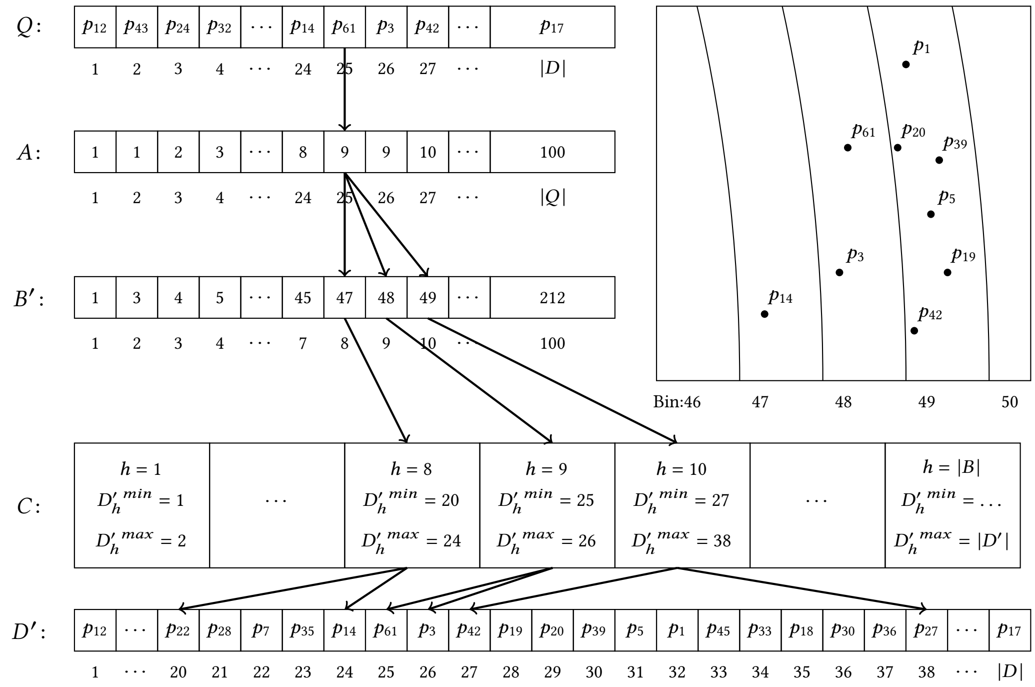

Q:
p_{12} p_{43} p_{24} p_{32} ··· p_{14} p_{61} p_3 p_{42} ··· p_{17}
1 2 3 4 ··· 24 25 26 27 ··· $|D|$
A:
1 1 2 3 ··· 8 9 9 10 ··· 100
1 2 3 4 ··· 24 25 26 27 ··· $|Q|$
B':
1 3 4 5 ··· 45 47 48 49 ··· 212
1 2 3 4 ··· 7 8 9 10 ··· 100
C:
$h = 1$ $D'^{min}_h = 1$ $D'^{max}_h = 2$
···
$h = 8$ $D'^{min}_h = 20$ $D'^{max}_h = 24$
$h = 9$ $D'^{min}_h = 25$ $D'^{max}_h = 26$
$h = 10$ $D'^{min}_h = 27$ $D'^{max}_h = 38$
···
$h = |B|$ $D'^{min}_h = \dots$ $D'^{max}_h = |D'|$
D':
p_{12} ··· p_{22} p_{28} p_7 p_{35} p_{14} p_{61} p_3 p_{42} p_{19} p_{20} p_{39} p_5 p_1 p_{45} p_{33} p_{18} p_{30} p_{36} p_{27} ··· p_{17}
1 ··· 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 ··· $|D|$
p_1 p_{61} p_{20} p_{39} p_5 p_3 p_{19} p_{14} p_{42}
Bin:46 47 48 49 50

Figure 4: Grid indexing example, where Q is the point array, A is the lookup array, B' is the unique bin array, C is the range array, and D' is the sorted data array. For clarity, only one reference point is shown. When there are multiple reference points, B' will be a multidimensional array, constructed as described in Section 4.2.

in B' or C, which is why indices of B' correspond to the indices of C. Bins 47, 48, and 49 contain the following candidate points and comprise the candidate set K, where $K = \{D'[20], \dots, D'[24]\} \cup \{D'[25], D'[26]\} \cup \{D'[27], \dots, D'[38]\}$. The Euclidean distance is computed between p_{61} and each point in its candidate set, K.

With multiple reference points the search only needs to consider more bins in B'. These are additional binary searches whose effects on the performance of COSS is discussed in Section 6.4.1.

4.4 Selecting the Location of Reference Points

We propose two reference point placement heuristics. While the selected position of each reference point in the coordinate space is arbitrary, the positions will impact the pruning efficiency of the algorithm. We note that finding the optimal positions of reference points that minimize the number of point comparisons is intractable.

RP-Inner: This strategy places multiple reference points close to the center of the data. We take the average value of the data in each dimension and place the first reference point at that location. The subsequent reference points are placed around the centered reference point in an expanding area. This creates a large number of small bins near the average center of the data, with bins that grow in size with distance from the center. Figure 5(a) shows an example of this placement strategy and the pattern of bins that develops near the center of the data.

RP-Outer: This strategy places the reference points at the edges of the data. The first reference point will be placed at the farthest range in every coordinate. To place the subsequent reference points, we take the number of dimensions, n, and divide that by the number of remaining reference points $v = n/(|W| - 1)$. The reference points will have v max range values, with the rest of their coordinate values being 0 (every reference point, besides the first, has v unique non-zero values). This scatters all of the reference points around the outside of the data distribution. Figure 5(b) shows what three reference points on the outskirts of the data looks like. The bins made by the expanding rings are fairly consistent in size. Note that as the distance from one reference point increases, the distances

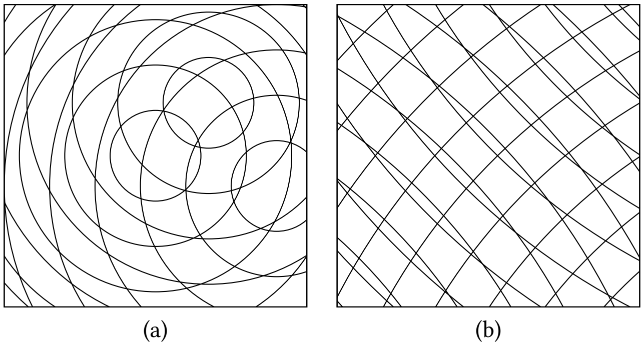


Figure 5: (a) shows the pattern generated with the RP-Inner placement strategy. (b) shows the pattern generated by the RP-Outer placement strategy.

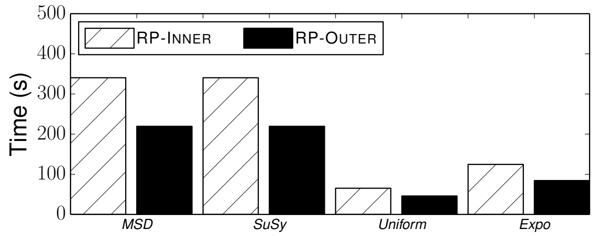


Figure 6: We compare the runtimes for RP-Inner and RP-Outer reference point placement strategies with *MSD* ($n = 90, \epsilon = 0.007$), *SuSy* ($n = 18, \epsilon = 0.015$), *Uniform* ($n = 10, \epsilon = 0.35$), *Expo* ($n = 16, \epsilon = 0.04$).

to the other reference points decrease. The larger bins from the increased distance are offset by the smaller bins from the decreased distances, creating bins with a more even point distribution than RP-Inner.

Placement Method Comparison: Figure 6 shows the difference in total runtime for two real-world and two synthetic datasets. In every experiment, the RP-Outer placement strategy outperforms the RP-Inner placement strategy. Therefore, in all following evaluations of COSS we use the RP-Outer placement strategy.

5 GPU ALGORITHM AND OPTIMIZATIONS

In this section we present an overview of COSS and algorithm optimizations.

5.1 Algorithm Overview

We present the pseudocode of COSS in Algorithm 1 and refer to the optimizations outlined later in this section. The COSSSelfJoin procedure begins by loading in the dataset D on line 2 and then ordering D according to the variance in each dimension (line 3 see Section 5.6). We then select our reference point placement based on the values in D' (line 4, see Section 4.4), set the number of threads per point (line 5, see Section 5.2), and construct our index (line 6, see Section 4.2). We compute the number of batches on line 7, initialize the max result size to zero (line 8) and then begin looping through every batch on line 9. For every batch we; execute COSSKernel on the GPU (line 10) as described on lines 17–30, check if the result size is smaller than the max results size (line 11) and pin memory for the result set buffer (line 12) if the result size was larger, and finally transfer and store the results on the host (lines 14 and 15).

The COSSKernel begins by storing the global thread id, and the query point's id and bin (lines 18 to 20). For every possible adjacent bin (line 21), we get the bin number to search (line 22) and search B' with a binary search to find which index that bin is at in B' (line 23). Note that on line 22, to avoid duplicate calculations by exploiting the reflexive property of the distance calculation, we apply the unidirectional comparison strategy [11] described in Section 4.3. If the bin is found in B' (line 24) we retrieve the min and max index into Q from C and store those points as the candidate set of the bin, Z (lines 25 and 26). The pseudocode refers to assigning a single thread to process one query point ($t = 1$). When $t > 1$ we divide the $|Z|$ candidate points to be processed by t threads on line 27. In particular, for each query point, let $l = 1, \ldots, t$. Thread l is assigned candidate point j where $(l - 1) \bmod j = 0$. Then we compute the distance between the query point and all candidate points, checking if they are within ϵ (lines 28 and 29). If the distance is within ϵ we add the point pair to the result set (line 30).

Algorithm 1 COSS Algorithm

```
 1: procedure COSSSelfJoin
 2:     $D \leftarrow$ inputData()
 3:     $D' \leftarrow$ dimensionalOrdering($D$)
 4:     $W \leftarrow$ placeReferencePoints($D'$)
 5:     $t \leftarrow$ setNumberThreadsPerPoint()
 6:     $Q, A, B', C, D' \leftarrow$ constructIndex($D', W$)
 7:     $g \leftarrow$ computeNumberOfBatches($D'$)
 8:     maxSize $\leftarrow 0$
 9:     for $i \in (1, 2, \ldots, g)$ do
10:         resultSize $\leftarrow$ COSSKernel($Q, A, B', C, D', t$)
11:         if resultSize > maxSize then
12:             pinMemory(resultSize)
13:             maxSize $\leftarrow$ resultSize
14:         results $\leftarrow$ tranferResultsToHost(resultSize)
15:         $R \leftarrow R \cup$ results
16:
17: procedure COSSKernel($Q, A, B', C, D', t$)
18:     tid $\leftarrow$ getThreadID()
19:     queryPointID $\leftarrow Q[tid/t]$
20:     queryPointBin $\leftarrow A[tid/t]$
21:     for $i \in (1, 2, \ldots, 3^{|W|})$ do
22:         binToSearch $\leftarrow$ generateNextBin(queryPointBin, $i$)
23:         binToSearchIndex $\leftarrow$ searchBins($B'$, binToSearch)
24:         if binToSearchIndex $\neq \emptyset$ then
25:             minIndex, maxIndex $\leftarrow C$[binToSearchIndex]
26:             $Z \leftarrow \{Q[minIndex], \ldots, Q[maxIndex]\}$
27:             for $j \in (1, 2, \ldots, |Z|)$ do
28:                 distance $\leftarrow dist(Q[tid/t], Q[Z[j]], D')$
29:                 if distance $\leq \epsilon$ then
30:                     results $\leftarrow$ results $\cup$ (queryPointID, $Z[j]$)
```

5.2 GPU Thread Allocation

Modern GPUs have thousands of cores. We can make use of the cores by dividing the distance calculations for a query point across multiple threads. (The query point q is the point that is evaluating a set of candidate points K found through the use of the index.) A query point q with an associated candidate set, K, will divide the work across multiple threads t. Where each thread has $|K|/t$ distance calculations to compute. Without the loss of generality and for illustrative purposes we assume t divides $|K|$. Figure 7 plots the runtime vs. the number of threads per a point on the *MSD* dataset (other datasets exhibit similar performance with the change in threads per a point). Every query point is assigned multiple threads to compute the distance calculations to refine the candidate set, K. From the experiments in Figure 7, we find that $t = 8$ achieves a good performance, and we use $t = 8$ threads in all the evaluation.

5.3 Batching Scheme

Depending on the search distance, ϵ, and data distribution, the result set size, $|R|$, may exceed global memory capacity. To ensure that the result set does not exceed global memory capacity, we divide the total computation into several batches. The batched execution allows us to concurrently execute tasks (e.g., pinning memory and host-GPU data transfers) in multiple CUDA streams. In this paper we use two CUDA streams.

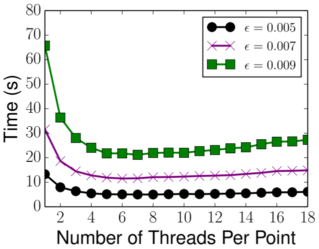


Figure 7: Runtime vs. the number of threads (t) on the *MSD* dataset ($n = 90, S = 4 - 1892$) shows that the while a small number of threads per a point has significantly longer runtime, $t = 5 - 10$ achieves good performance.

The number of query points per a batch, h, is governed by the amount of global memory available to store results, which is contingent on the selectivity discussed in Section 3.1. The CUDA block size b and the number of blocks p determine the number of points that are evaluated in each batch/kernel invocation. We experimentally found that a block size $b = 1024$ yields the best performance. We use the number of blocks p per kernel invocation to determine h. The number of points evaluated per a batch is $h = b \cdot p$ and we can use h to find the total number of batches $g = \lceil |D|/h \rceil$. The total number of threads per a batch is $u = bpt$, where t is the number of threads per a point as discussed in Section 5.2.

5.4 Concurrent Execution of Batches

The result sets generated on the GPU are large, it is more efficient to manually pin the memory needed and then reuse the pinned memory buffer. By pinning memory we can increase the effective bandwidth of the PCIe interconnect that connects the GPU to the host [26]. To determine how much memory needs to be pinned, each CUDA stream (COSS is evaluated with 2 streams) will execute one batch then take the size of the results and pin that much memory for the stream. The subsequent batches with smaller result set sizes can reuse the pinned memory buffer. After every batch is computed by a kernel invocation on the GPU we ensure that the pinned memory is sufficiently large to store the data, if not, we reallocate a larger pinned memory buffer.

High-dimensional distance similarity searches are compute bound. We can take advantage of the high computation time to hide memory transfers. Using two concurrent streams, one stream executes the kernel, and one sends results back to the host. Other host-side tasks are mostly hidden using two streams.

5.5 Short Circuiting the Distance Calculations

The distance calculation as defined in Section 2, is the summation of the distances in individual coordinate dimensions. When calculating this distance we compute the first term of the summation and add it to a running total distance, then compute and add the second term to the running total and so forth for all n terms. After we compute and add each term, we can check to see if the running total has exceeded the distance threshold, ϵ. If the running total exceeds ϵ we stop computing the distance, which reduces the total number of floating point operations computed.

5.6 Dimensional Ordering

We can increase the effectiveness of short circuiting (Section 5.5) by finding the variance of each dimension of the original data. We can then rearrange the point coordinates of the data so that the highest variance is first and the lowest variance is last. For example, the points $p_i \in D$ where $p_i = (x_1, x_2, \dots, x_n)$, where n is the number of dimensions, will become $p_i = (x_{n_{max}}, x_{n_{max}-1}, \dots, x_{n_{min}})$, where n_{max} is the dimension that had the most variance and n_{min} is the dimension that had the least amount of variance. When computing the distance calculations, this will result in the distance accumulating faster, leading to an earlier short circuit for most points. In high dimensions, this is especially effective because of the large number of dimensions and how early a distance calculation can exceed ϵ. Other grid-based algorithms use a similar dimensionality reordering method, including the two reference implementations, GPU-Join and Super-EGO.

Table 1: Datasets used in the evaluation.

Dataset	n	Size ($\|D\|$)	ϵ	Selectivity (S)
MSD [6]	90	515, 345	0.005 − 0.01	4 − 1892
SuSy [2]	18	5×10^6	0.01 − 0.02	5 − 780
Higgs [2]	28	11×10^6	0.035 − 0.045	5 − 91
Uniform	10	2×10^6	0.25 − 0.45	2 − 551
Expo	16	2×10^6	0.03 − 0.05	4 − 1226

6 EXPERIMENTAL EVALUATION

6.1 Experimental Methodology

All host code is written in C/C++ and GPU code is written in CUDA and is compiled with the GNU compiler with the O3 optimization flag. GPU code is compiled using CUDA 9. Our platform consists of 2x Intel Xeon E5-2620 v4 CPUs clocked at 2.10 GHz, with a total of 16 physical cores, and 128 GiB of main memory, equipped with an Nvidia GP100 GPU with 16 GiB of global memory (Pascal generation).

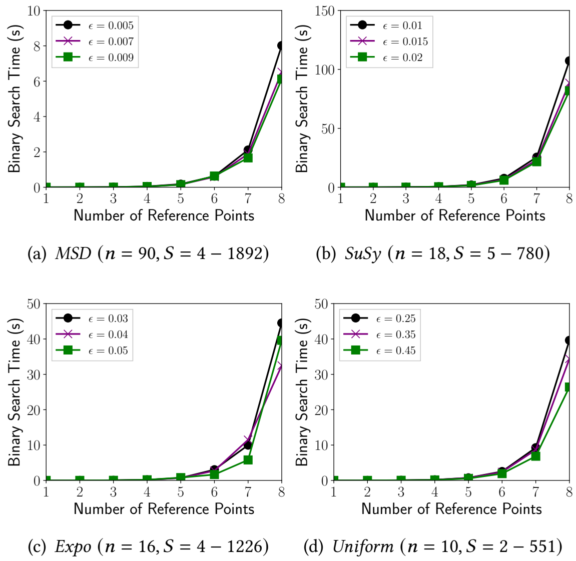
In all experiments, we report the average runtime as averaged over 3 trials. As described in Section 6.3, we compare our algorithm, COSS, to GPU-Join, and Super-EGO. We refer to the total runtime of each algorithm using respective algorithm components described in Section 6.3.

To ensure that our experiments reflect real-world application scenarios, we report the selectivity of our searches as defined in Section 3.1.

6.2 Datasets

We select three real-world datasets from the literature and generate two synthetic datasets. *MSD* is a 90-D dataset containing song features, *SuSy* (18-D) and *Higgs* (28-D) are from particle physics. All datasets used in this paper are normalized for each dimension in the range [0, 1]. The range of dataset dimensions is consistent with other papers (real-world datasets in other works span $n = 9$−32 [21], $n = 2 - 784$ [19], and $n = 18 - 90$ [12]).

We selected uniformly and exponentially distributed datasets. The *Uniform* dataset represents the case where indexing on the data point coordinates leads to an increasingly exhaustive search (Section 3.1). The *Expo* dataset represents the opposite of the *Uniform* dataset, where there is one over-dense region and a large under-dense region. *Expo* was generated with $\lambda = 40$. The datasets are summarized in Table 1.



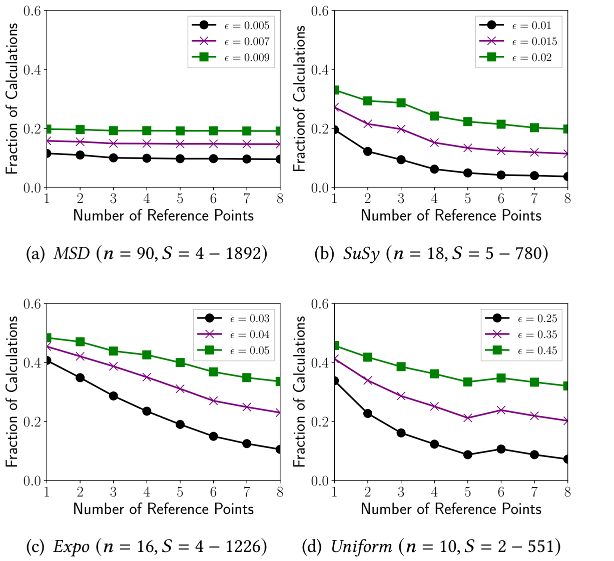
(a) *MSD* ($n = 90, S = 4 - 1892$)

(b) *SuSy* ($n = 18, S = 5 - 780$)

(c) *Expo* ($n = 16, S = 4 - 1226$)

(d) *Uniform* ($n = 10, S = 2 - 551$)

Figure 8: Index binary search time (s) vs. number of reference points.



(a) *MSD* ($n = 90, S = 4 - 1892$)

(b) *SuSy* ($n = 18, S = 5 - 780$)

(c) *Expo* ($n = 16, S = 4 - 1226$)

(d) *Uniform* ($n = 10, S = 2 - 551$)

Figure 9: Fraction of distance calculations vs. number of reference points.

6.3 Implementations

COSS: COSS is evaluated when we enable all of the optimizations outlined in Section 5. We select a fixed set of parameters that achieve good performance across all experimental scenarios. COSS is configured with 2 CUDA streams, $t = 8$ threads per a point, 6 reference points and the RP-Outer reference point placement strategy (Section 4.4). We include the time it takes to construct the index, pin and transfer memory, perform the distance calculations, and store the final results on the host. COSS is evaluated using 64-bit floating point values.

GPU-Join (GPU Reference Implementation): As described in Section 3.3.2, GPU-Join [12] uses a grid-based indexing scheme for the GPU. The algorithm uses several optimizations, including projecting the coordinates into $k < n$ dimensions, reordering the data by variance in each dimension, short circuiting the distance calculation, and reducing distance calculations by searching on an un-indexed dimension. We use the experimental parameters and configuration used in Gowanlock and Karsin [12] when executing GPU-Join. In particular, we enable all of their optimizations, and index on $k = 6$ dimensions, and use 256 threads per block. GPU-Join is executed using 64-bit floating point values which is consistent with COSS. Using the experimental methodology in Gowanlock and Karsin [12], the runtime excludes the time to index the dataset, but includes all GPU computation, and transferring the data and results to and from the GPU.

In contrast to COSS, GPU-Join does not eliminate duplicate searches for the same point, as GPU-Join [12] presents performance results that are directly applicable to both the self-join and the semi-join on two datasets (the self-join can eliminate duplicate work, but the semi-join on two datasets cannot). Therefore, we expect that GPU-Join will perform at least double the distance calculations as COSS.

Super-EGO (CPU Reference Implementation): As described in Section 3.3.1, Super-EGO indexes using a grid with ϵ-length cells, and prunes the search by employing a data reordering scheme. The algorithm is parallelized for multi-core CPUs. We execute Super-EGO using 16 threads (the number of physical cores on our platform). Since Super-EGO fails to execute when using 64-bit floating point values, we execute the algorithm with 32-bit values. This gives an advantage to Super-EGO over GPU-Join and COSS. The runtime is computed as the time to EGO-sort and join. The code is publicly available on the author's website.[1]

6.4 Impact of COSS Parameters on Performance

Performance is evaluated on all datasets in this subsection except *Higgs* which was omitted due to space constraints. The results from *Higgs* are consistent with the datasets used in this subsection.

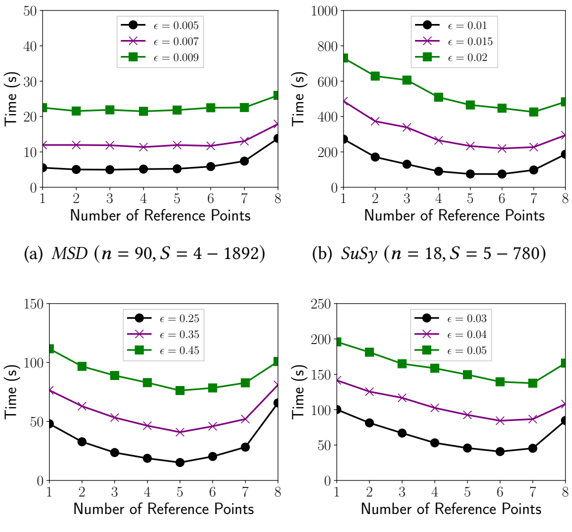
6.4.1 *Binary Search Time.*

COSS uses binary searches to find adjacent non-empty bins in B'. We evaluate the binary search time vs. number of reference points and plot it in Figure 8. While the binary search times are insignificant in small numbers, when the number of reference points increases beyond 6 ,with $3^6/2$ searches per point (see Section 4.3), it begins to impact performance. In Figure 8 we observe the exponential growth in search time with the increase in the number of reference points. From this we conclude that it would be disadvantageous to use $\gtrsim$ 6 reference points.

6.4.2 *Pruning Efficiency.*

The efficiency of COSS is dependent on the amount of pruning that it can achieve. The amount of pruning is dependent on both the reference point placement strategy and number of reference points. Figure 9 plots the fraction of distance

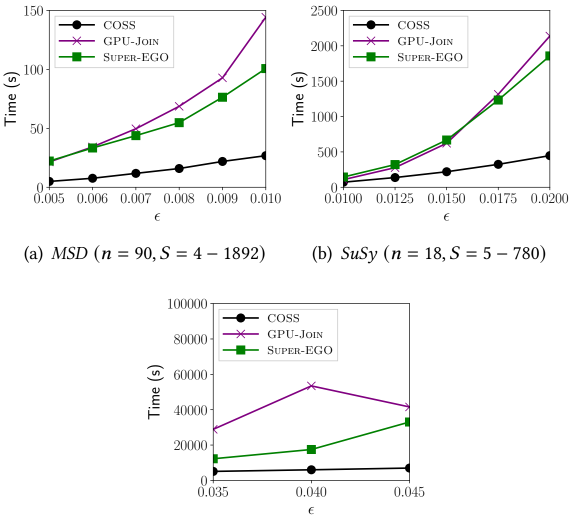
[1] https://www.ics.uci.edu/~dvk/code/SuperEGO.html.



(a) *MSD* $(n = 90, S = 4 - 1892)$ (b) *SuSy* $(n = 18, S = 5 - 780)$

(c) *Expo* $(n = 16, S = 4 - 1226)$ (d) *Uniform* $(n = 10, S = 2 - 551)$

Figure 10: Runtime (s) vs. number of reference points.



(a) *MSD* $(n = 90, S = 4 - 1892)$ (b) *SuSy* $(n = 18, S = 5 - 780)$

(c) *Higgs* $(n = 28, S = 5 - 91)$

Figure 11: Runtime (s) vs. ϵ on real-world datasets. Comparing COSS, GPU-Join, and Super-EGO.

calculations vs. number of reference points where the fractions is calculated as $|K|/|D|^2$, where $|K|$ is the number of distance calculations made by COSS. Increasing the number of reference points greatly reduces the total number of distance calculations.

6.4.3 Effect of Number of Reference Points on Runtime. From the previous experiments we can see that the number of reference points impacts the performance significantly. Figure 10 shows the response time based on the number of reference points used. Figure 8 combined with Figure 9 explains how the response time in Figure 10 increases after 6 reference points. While the percentage of distance calculations decrease, the number of binary searches increases rapidly. There is a trade off between time spent on the searches and time spent on the distance calculations. We find that 6 reference points performs well on all experimental scenarios.

6.5 Comparison to Reference Implementations

In this section we look at the experimental results across three real world data sets and one synthetic dataset. Table 1 shows a summary of the datasets used. We choose to use 6 reference points for making comparisons to other methods to maintain consistency across different datasets.

6.5.1 Real World Datasets. The real-world datasets (*MSD*, *SuSy* and *Higgs*) are used to compare the performance of COSS with Super-EGO and GPU-Join. The datasets dimensions' span $n = 18 - 90$ and are evaluated on a large range of search distances and selectivity values.

***MSD* Dataset:** Figure 11(a) shows the runtime vs. ϵ on the *MSD* dataset. The performance of COSS degrades gracefully with increasing ϵ. COSS significantly outperforms the reference implementations. We find that COSS has a speedup of up to 5.38× and 3.76× over GPU-Join and Super-EGO, respectively.

***SuSy* Dataset:** In Figure 11(b) we plot the runtime vs ϵ on the *SuSy* dataset. From the plot we observe that while all three methods (COSS, GPU-Join, and Super-EGO) have similar runtimes at $\epsilon = 0.01$, both GPU-Join and Super-EGO suffer a rapid increase in runtime with the increasing ϵ values. COSS yields a speedup of up to 4.78× over GPU-Join and 4.15× over Super-EGO.

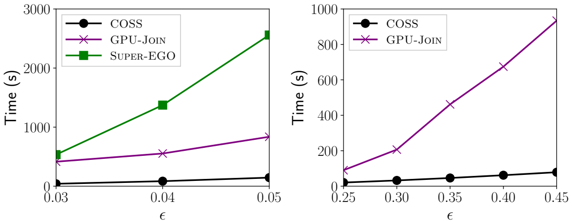
***Higgs* Dataset:** Figure 11(c) plots the runtime vs ϵ on the *Higgs* dataset. We observe that COSS has better performance at all ϵ values with a speedup of up to 8.85× over GPU-Join and 4.73× over Super-EGO.

6.5.2 Synthetic Datasets. On the synthetic datasets (*Expo* and *Uniform*), the data has the same variance in each dimension. Therefore, all three algorithms are unable to use their respective optimizations that exploit the statistical properties of the data (e.g., dimensional ordering in Section 5.6).

Exponentially distributed data allows the self-join to find a reasonable number of neighbors with a moderate search radius. Whereas uniformly distributed data requires a large search radius to find many neighboring points (Section 3.1). The selectivity yielded by *Expo* is more similar to real-world data distributions than *Uniform.*

Exponentially Distributed Data: Figure 12(a) plots the runtime vs. ϵ on the *Expo* datasets. From the figure, we observe that COSS significantly outperforms both GPU-Join and Super-EGO. For example, at $\epsilon = 0.05$, we obtain a speedup of 5.78× and 17.69×, over GPU-Join and Super-EGO, respectively.

Grid Killer – Uniformly Distributed Data: Figure 12(b) plots the runtime vs. ϵ on the *Uniform* dataset. Note that Super-EGO failed to execute on this dataset. COSS achieves a speedup of 11.8× over GPU-Join at $\epsilon = 0.45$. From the figure we observe that while the performance of COSS degrades gracefully with increasing ϵ, the pruning efficiency of GPU-Join decreases rapidly.



(a) *Expo* ($n = 16, S = 4 - 1226$) (b) *Uniform* ($n = 10, S = 2 - 551$)

Figure 12: Runtime (s) vs. ϵ on synthetic datasets. Comparing COSS, GPU-Join, and Super-EGO.

Table 2: Average speedup of COSS over GPU-Join and Super-EGO across all values of ϵ in Section 6.5.

Dataset	*MSD*	*SuSy*	*Higgs*	*Expo*	*Uniform*
GPU-Join	4.50	3.04	6.84	7.49	8.79
Super-EGO	3.88	3.08	3.35	15.66	-

As described in Section 3.1 on uniformly distributed datasets, to achieve a reasonable average number of neighbors per point, ϵ needs to be sufficiently large. In Figure 11(b), GPU-Join has 4, 4, 3, 3, and 3 cells in each indexed dimension at $\epsilon = 0.25$, 0.30, 0.35, 0.40, and 0.45, respectively. Consequently, the grid used in GPU-Join is unable to prune a large fraction of the points, and the algorithm approaches the brute force quadratic complexity. For example, in the worst case, if there are 3 cells in each dimension, then a point located in the center of the grid is compared to all $|D|$ points in the dataset. Similarly, all multidimensional data access methods that directly index on the coordinates of the input data will suffer from the curse of dimensionality.

As discussed in Kalashnikov [19] (Super-EGO), when the search distance exceeds half of the bounding volume, the algorithm degrades to brute force. While optimizations such as short circuiting the distance calculation are able to reduce point comparison cost, only a better pruning strategy, such as that employed by COSS, is able to significantly improve performance.

7 DISCUSSION AND CONCLUSIONS

In this paper, we propose COSS, a GPU-efficient coordinate-oblivious similarity self-join algorithm. To our knowledge, no other indexing methods have been proposed that utilize distance space for the GPU. We summarize the performance of COSS in Table 2 which plots the average speedup obtained on all datasets in Table 1. This shows that our novel index mitigates the curse of dimensionality problem on datasets up to 90 dimensions. While the reference implementations degrade to brute force searches on uniformly distributed data, COSS is still able to prune the search in this scenario. Overall, our index significantly outperforms the two reference implementations which index on the coordinate space.

Future work includes transforming coordinate space into distance space for other related similarity search problems, such as k-nearest neighbor searches. While we proposed two heuristics for reference point placement in this paper, a future direction is to investigate other placement strategies.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. 1849559. This work has been supported by the Arizona Board of Regents, Regents' Innovation Fund.

REFERENCES

[1] Daichi Amagata, Takahiro Hara, and Chuan Xiao. 2019. Dynamic Set kNN Self-Join. In *IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 818–829.

[2] Pierre Baldi, Peter Sadowski, and Daniel Whiteson. 2014. Searching for exotic particles in high-energy physics with deep learning. *Nature communications* 5 (2014), 4308.

[3] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. 1990. The R*-tree: an efficient and robust access method for points and rectangles. In *Proceedings of the ACM SIGMOD international conference on Management of data*. 322–331.

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

[5] S Berchtold, DA Keim, and HP Kriegel. 2001. The X-Tree: An index structure for high-dimensional data. *Readings in multimedia computing and networking* 451 (2001), 28–39.

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

[7] 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. *ACM SIGMOD Record* 30, 2 (2001), 379–388.

[8] Přemysl Čech, Jakub Lokoč, and Yasin N Silva. 2020. Pivot-based approximate k-NN similarity joins for big high-dimensional data. *Information Systems* 87 (2020), 101410.

[9] Kaushik Chakrabarti and Sharad Mehrotra. 1999. The hybrid tree: An index structure for high dimensional feature spaces. In *Proceedings of the 15th International Conference on Data Engineering*. IEEE, 440–447.

[10] Yilin Feng, Jie Tang, Meilin Liu, Chongjun Wang, and Junyuan Xie. 2018. Fast Document Cosine Similarity Self-Join on GPUs. In *2018 IEEE 30th International Conference on Tools with Artificial Intelligence*. IEEE, 205–212.

[11] Michael Gowanlock and Ben Karsin. 2019. Accelerating the similarity self-join using the GPU. *Journal of parallel and distributed computing* 133 (2019), 107–123.

[12] Michael Gowanlock and Ben Karsin. 2019. GPU-Accelerated Similarity Self-Join for Multi-Dimensional Data. In *Proceedings of the 15th International Workshop on Data Management on New Hardware*. 1–9.

[13] Michael Greenspan and Mike Yurick. 2003. Approximate kd tree search for efficient ICP. In *Proceedings of the Fourth International Conference on 3-D Digital Imaging and Modeling*. IEEE, 442–448.

[14] Antonin Guttman. 1984. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD international conference on Management of data*. 47–57.

[15] Joseph M Hellerstein and Avi Pfeffer. 1994. *The RD-tree: An index structure for sets.* Technical Report. University of Wisconsin-Madison Department of Computer Sciences.

[16] Yun-Wu Huang, Ning Jing, Elke A Rundensteiner, et al. 1997. Spatial joins using R-trees: Breadth-first traversal with global optimizations. In *VLDB*, Vol. 97. Citeseer, 25–29.

[17] Edwin H Jacox and Hanan Samet. 2007. Spatial join techniques. *ACM Transactions on Database Systems (TODS)* 32, 1, Article 7 (2007).

[18] Hosagrahar V Jagadish, Beng Chin Ooi, Kian-Lee Tan, Cui Yu, and Rui Zhang. 2005. iDistance: An adaptive B+-tree based indexing method for nearest neighbor search. *ACM Transactions on Database Systems (TODS)* 30, 2 (2005), 364–397.

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

[20] Jinwoong Kim, Sul-Gi Kim, and Beomseok Nam. 2013. Parallel multi-dimensional range query processing with R-trees on GPU. *J. Parallel and Distrib. Comput.* 73, 8 (2013), 1195–1207.

[21] Michael D Lieberman, Jagan Sankaranarayanan, and Hanan Samet. 2008. A fast similarity join algorithm using graphics processing units. In *Proceedings of the 24th IEEE International Conference on Data Engineering*. IEEE, 1111–1120.

[22] Youzhong Ma, Ruiling Zhang, Shijie Jia, Yongxin Zhang, and Xiaofeng Meng. 2019. An efficient similarity join approach on large-scale high-dimensional data using random projection. *Concurrency and Computation: Practice and Experience* 31, 20 (2019), e5303.

[23] Marius Muja and David G Lowe. 2009. Fast approximate nearest neighbors with automatic algorithm configuration. *VISAPP (1)* 2, 331-340 (2009), 2.

[24] Sameer A Nene and Shree K Nayar. 1997. A simple algorithm for nearest neighbor search in high dimensions. *IEEE Transactions on pattern analysis and machine intelligence* 19, 9 (1997), 989–1003.

[25] NVIDIA. 2017. *P100 The Most Advanced Data Center Accelerator Ever Built. Featuring Pascal GP100, the World's Fastest GPU.* Retrieved January 31, 2020 from https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/tesla-p100/pdf/nvidia-tesla-p100-datasheet.pdf

[26] NVIDIA. 2018. *Pascal Tuning Guide.* Retrieved January 31, 2020 from http://docs.nvidia.com/cuda/pascal-tuning-guide/index.html

[27] Martin Perdacher, Claudia Plant, and Christian Böhm. 2019. Cache-oblivious high-performance similarity join. In *Proceedings of the International Conference on Management of Data*. 87–104.

[28] Sushil K Prasad, Michael McDermott, Xi He, and Satish Puri. 2015. GPU-based Parallel R-tree Construction and Querying. In *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*. IEEE, 618–627.

[29] DA Rachkovskij. 2019. Fast Similarity Search for Graphs by Edit Distance. *Cybernetics and Systems Analysis* 55, 6 (2019), 1039–1051.

[30] Chuitian Rong, Xiaohai Cheng, Ziliang Chen, and Na Huo. 2019. Similarity joins for high-dimensional data using Spark. *Concurrency and Computation: Practice and Experience* 31, 20 (2019), e5339.

[31] Erich Schubert, Jörg Sander, Martin Ester, Hans Peter Kriegel, and Xiaowei Xu. 2017. DBSCAN revisited, revisited: why and how you should (still) use DBSCAN. *ACM Transactions on Database Systems (TODS)* 42, 3 (2017), 1–21.

[32] David A White and Ramesh Jain. 1996. Similarity indexing with the SS-tree. In *Proceedings of the Twelfth International Conference on Data Engineering*. IEEE, 516–523.

[33] Svante Wold, Kim Esbensen, and Paul Geladi. 1987. Principal component analysis. *Chemometrics and intelligent laboratory systems* 2, 1-3 (1987), 37–52.

[34] Ren Wu, Bin Zhang, and Meichun Hsu. 2009. Clustering billions of data points using GPUs. In *Proceedings of the combined workshops on UnConventional high performance computing workshop plus memory access workshop*. 1–6.

[35] Simin You, Jianting Zhang, and Le Gruenwald. 2013. Parallel spatial query processing on GPUs using R-trees. In *Proceedings of the 2nd ACM SIGSPATIAL International Workshop on Analytics for Big Geospatial Data*. 23–31.

[36] Kun Zhou, Qiming Hou, Rui Wang, and Baining Guo. 2008. Real-time kd-tree construction on graphics hardware. *ACM Transactions on Graphics (TOG)* 27, 5 (2008), 1–11.