GPU Accelerated Self-join for the Distance Similarity Metric

Michael Gowanlock
School of Informatics, Computing, & Cyber Systems
Northern Arizona University
Flagstaff, AZ, 86011
michael.gowanlock@nau.edu

Ben Karsin
Department of Information and Computer Sciences
University of Hawaii at Manoa
Honolulu, HI, 96822
karsin@hawaii.edu

***Abstract*—The self-join finds all objects in a dataset within a threshold of each other defined by a similarity metric. As such, the self-join is a building block for the field of databases and data mining, and is employed in Big Data applications. In this paper, we advance a GPU-efficient algorithm for the similarity self-join that uses the Euclidean distance metric. The search-and-refine strategy is an efficient approach for low dimensionality datasets, as index searches degrade with increasing dimension (i.e., the curse of dimensionality). Thus, we target the low dimensionality problem, and compare our GPU self-join to a search-and-refine implementation, and a state-of-the-art parallel algorithm. In low dimensionality, there are several unique challenges associated with efficiently solving the self-join problem on the GPU. Low dimensional data often results in higher data densities, causing a significant number of distance calculations and a large result set. As dimensionality increases, index searches become increasingly exhaustive, forming a performance bottleneck. We advance several techniques to overcome these challenges using the GPU. The techniques we propose include a GPU-efficient index that employs a bounded search, a batching scheme to accommodate large result set sizes, and a reduction in distance calculations through duplicate search removal. Our GPU self-join outperforms both search-and-refine and state-of-the-art algorithms.**

***Index Terms*—GPGPU, In-memory database, Query optimization, Self-join.**

I. Introduction

The similarity self-join is a routine operation, and is described as follows: given a dataset of objects, find all objects that have common attributes based on a similarity metric. In spatial applications, the problem typically focuses on distance metrics to find points that are near each other. However, in principle, distance similarity is a common metric, and can be applied to non-spatial applications as well. We focus on the distance similarity self-join that finds all points that are within a distance ϵ of each other using Euclidean distance.

Similarity self-joins are building blocks of existing data analysis algorithms. For example, the DBSCAN clustering algorithm requires range queries that search the neighborhood of all data points to find those within a given distance [1]. Other algorithms require range queries that constitute similarity joins, such as mining spatial association rules [2], information retrieval optimization [3], the OPTICS algorithm [4], and time series data analysis [5]. Therefore, the self-join is a fundamental component of established data analysis methods [6], [7] and, as such, is used in many Big Data applications.

Self-joins and the related similarity join typically target either low or high dimensionality in the literature. This is because the methods used for the self-join in low dimensionality are often unsuitable for high dimensional data, and vice versa. As described in [8], a typical approach in low-dimensionality is to use the search-and-refine strategy as follows: *search* an index for points that may be within the search radius of a query point, which generates a candidate set, and then *refine* these points by performing the distance calculation between the query point and all points in the candidate set. This technique has been shown to be very efficient for low dimensional data; however, index performance degrades with dimensionality. We consider data up to 6-D, which is within the regime where index search performance degradation is not prohibitive for canonical spatial indexing schemes, and thus allows us to make a direct comparison to the search-and-refine approach.

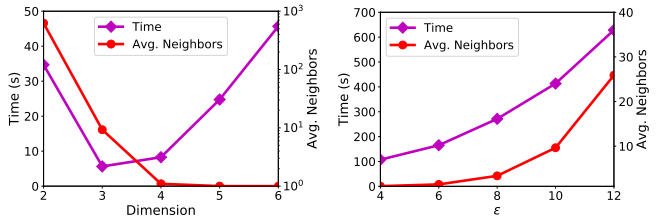


Fig. 1: Problem overview: (a) Computing the distance similarity self-join on 2 million data points in 2–6 dimensions with the distance $\epsilon = 1$ using an R-tree index, where each dimension is of equal length and the points are uniformly distributed in a (hyper)cube. (b) Time vs. ϵ for the 6-D dataset shown in (a). Datasets (*Syn-*) are described in Section VI-A.

We elaborate on performance as a function of dimensionality. Figure 1 (a) plots the self-join response time and the average number of neighbors per point vs. dimension for datasets with 2 million points that are indexed in the R-tree [9] on the CPU. Since the number of data points are kept constant, as the dimensionality increases, the data density decreases, as the average distance between objects increases with dimension [10]. Therefore, the average number of neighbors per point decreases significantly with dimensionality. While this

demonstration is not representative of all possible scenarios, it shows two interesting computational problems. First, from Figure 1 (a), we see that the greatest response time (i.e., worst performance) occurs at 2 and 6 dimensions. In 2-D, there are many neighbors to consider per point, and while the R-tree index filters many of them, we must still perform costly Euclidean distance calculations between a large number of points to find those within the ϵ query distance. Second, at 6-D there are very few neighbors within ϵ per point, unless ϵ is increased (Figure 1 (b)). However, the index search for 6-D data is more exhaustive in comparison to 2-D. Thus, index performance degenerates with increasing dimension, caused by the well-known curse of dimensionality [11]–[14].

In this paper, we propose an algorithm that leverages GPU hardware to efficiently compute the distance similarity self-join. This paper makes the following contributions:

- We present an efficient indexing strategy that bounds the search for nearby points and exploits the high memory bandwidth and parallelism afforded by the GPU.
- By carefully considering the domain decomposition provided by the index, we provide a selection strategy that eliminates duplicate searches and distance calculations. We show that the performance of this optimization is dependent on GPU cache utilization.
- For self-joins that yield a large number of nearby neighbors which would exceed the memory capacity of the GPU, we exploit an efficient batching scheme between host and GPU to incrementally compute the entire self-join result.
- We show that for a fixed dimension, our approach achieves similar performance gains on real-world and uniformly distributed synthetic datasets.
- Related to the above, we motivate that uniformly distributed data should represent a worst-case scenario for our grid-based indexing approach. Despite this, our approach is still significantly faster than the reference implementation using an R-tree across the 2–6 dimensions evaluated. Furthermore, our approach outperforms the parallel state-of-the-art implementation across nearly all experimental scenarios.

The paper is outlined as follows: Section II provides background information, Section III formalizes the problem, Section IV presents our GPU indexing strategy, Section V describes additional techniques used to improve self-join performance, Section VI outlines the methodology and experimental evaluation, and finally, we conclude the work in Section VII.

II. Background

We consider several relevant categories of related work. The self-join problem is a special case of a join operation on two different sets of data points (or feature vectors). It is also similar to the problem of querying a database to find the subset of points whose values are within an ϵ distance from a query point. Therefore, works in these other areas are directly applicable to the self-join problem. All of these operations are typically supported by indexing data structures that are used to accelerate range queries on the input dataset. Thus, since we use GPUs to improve performance in this work, we also consider related work that focuses on algorithmic transformations and GPU optimized indexes. We present an overview of each of these relevant areas below.

Search-and-refine: Several works have studied the similarity-join problem on the CPU [3], [6], [8], [15], [16]. We focus on [6] as it uses the canonical search-and-refine strategy to compute the similarity self-join to accelerate clustering by calculating the neighbors of each point before clustering.

The authors in [6] utilize the search-and-refine strategy and evaluate their approach using the R*-Tree [17] and X-tree [18] to accelerate multidimensional searches on both 9-D and 64-D datasets. They report that performing the self-join first, instead of a series of individual range queries in the instruction flow of the clustering algorithm can significantly improve clustering performance. Using the self-join over the iterative approach, they achieve significant performance gains in the context of out-of-core processing, which has different overheads than the in-memory processing examined here. This work shows that the self-join is used in other algorithms, and that indexes improve self-join performance.

Employing Grids and Grid-based Indexes: Spatial indexes, such as the R*-Tree [17] mentioned above, are highly efficient at pruning the search space because the index is constructed based on the input data. Thus, regions with high data densities generate more data partitions (fewer partitions are needed in low density regions). This is similar to many other tree-based indexes such as k-d trees [19] or quad trees [20].

In contrast to these index-trees, grid-based methods make a static partitioning of the data, whereby some data partitions may contain over-densities of the data and other partitions may contain mostly empty space. Thus, the index is independent of the data distribution. One drawback of grids is that index-trees may be much faster at data retrieval on data distributions for which they are designed. Consequently, data-dependent index-trees have been the predominant index type for the search-and-refine strategy above. However, a benefit of grids is that they are fast to construct, and can be very space efficient, as their size is predominantly independent of the data distribution.

Grid-based approaches have been used to solve the similarity join problem on the CPU. The "epsilon grid order" [21] overlays the physical space with a non-materialized n-dimensional grid, where each grid cell is of length ϵ (the query distance). To summarize their approach, finding points within ϵ is bounded to the points in the adjacent grid cells of the cell of a query point, and these cells can be pruned by determining how far away they lie based on the individual cell's n dimensional coordinates. Similarly, [8] advances an epsilon grid order approach (Super-EGO) that uses the data distribution to inform pruning the search, and is state-of-the-art for join operations. We compare our approaches to a multi-threaded implementation of Super-EGO in [8].

Indexes for the GPU: To reduce the number of distance calculations, several papers have advanced GPU-efficient indexes [7], [14], [22]–[26]. A major question in this field is whether indexes suited for the CPU are efficient when

implemented for the GPU. Since tree-based indexes require many branch instructions, a loss in parallel efficiency can occur on the GPU due to the SIMD architecture [27]. A key insight from [26] is that it is preferable to have a less selective index that is very efficient on the GPU, rather than a more selective (tree-based) index that suffers from additional overheads and lower parallel efficiency. Consequently, we will motivate the use of a non-tree-based indexing scheme. A hierarchical grid index for the GPU that is designed for efficient searches in skewed data distributions is presented in [22]. In our evaluation, we utilize both real-world and uniformly distributed data. However, future work includes examining skewed data in greater detail.

GPU Similarity Joins: The similarity-join in [7] uses a directory structure to reduce the number of candidates that may be within ϵ of a query point. On 8-D datasets of up to 8 million points, the GPU outperforms the CPU algorithm when ϵ is selected such that each point has 1–2 average neighbors [7]. Given the input data and result set sizes, the self-join may not have exceeded the GPU's global memory capacity. We use batching to enable results that exceed GPU memory capacity.

LSS [28] advances a GPU similarity join for high-dimensional data that achieves impressive results. However, the target dimensionality is higher than that considered here.

III. Problem Statement

The distance similarity self-join problem is described as follows. Let D be a database of points. Each point is denoted as p_i, where $i = 1, \ldots, |D|$. Each $p_i \in D$ has coordinates in n-dimensions, where each coordinate is denoted as x_j where $j = 1, \ldots, n$, and n is the number of dimensions of the point/feature vector. We use the following notation to denote a data point and associated coordinates: $p_i = (x_1, x_2, \ldots, x_n)$.

We find all points, $p_i \in D$, that are within the Euclidean distance, ϵ, of each other (known as a range or distance query). As an example, given points $a \in D$ and $b \in D$, we say that the points are within ϵ when the distance function, $dist(a, b) \leq \epsilon$, where $dist(a, b) = \sqrt{\sum_{j=1}^{n}(a(x_j) - b(x_j))^2}$. All processing occurs in-memory on the GPU and CPU.

IV. GPU Indexing

A. Motivation: Utilizing a Grid

There are a number of challenges that need to be overcome for efficient indexing on the GPU. Due to the constrained global memory capacity, it is crucial that the index structure is small. However, spatial index structures often encapsulate the entire space, and thus may retain information regarding empty space. While this may be feasible in low dimensionality (i.e., 2-D [29]), it is intractable in higher dimensions. In contrast to previous work in [29], we do not index empty cells.

Index-trees have been implemented for both the CPU [9] and GPU [14]. However, the non-deterministic nature of tree-traversals leads to branch divergence, which is known to degrade GPU performance due to thread serialization [27]. Also, the access patterns of such tree-traversals may lead to poor cache efficiency. In contrast, a search for nearby points in a grid can be bounded to adjacent cells (as discussed in the related work). Since the search is bounded, it is likely to have more regular memory access patterns in comparison to index-trees. As has been shown in other contexts, algorithms with regular memory access patterns on GPUs often yield significant performance gains over multi-core CPU algorithms [30].

Because the grid is constructed independently of the data distribution (with the exception of encompassing the entire dataset in each dimension), the grid is well-suited for employing a work-avoidance strategy. As we demonstrate, employing a grid enables us to reduce the number of distance calculations and index searches by a factor of $\sim$2.

We utilize a grid-based index for the GPU because: (i) the memory requirements of the index are small, which is important to maximize the space for other data; (ii) bounded grid index searches have regular memory access patterns and thus less thread divergence, which would otherwise degrade performance; and, (iii) the grid index presents an opportunity to reduce the total workload of the self-join.

B. Index Properties

In this section, where appropriate, we use similar notation as previous work [29] to describe our index. We denote g_j as the properties of the index structure in the j^{th} dimension. We find the minimum and maximum values of each dimension across all points. Using these values, we define as g_j^{min} and g_j^{max} as the minimum and maximum dimensions of our index grid, respectively. This defines the index range [g_j^{min}, g_j^{max}] where $g_j^{min} = \min_{p_i \in D} x_j - \epsilon$ and $g_j^{max} = \max_{p_i \in D} x_j + \epsilon$, for each dimension j. The range is appended by ϵ to avoid boundary conditions in grid cell lookup calculations.

To generate an n-dimensional grid, we compute grid cells in each dimension, where the length of each cell is ϵ. The number of cells in each dimension, $|g_j|$, is computed as $|g_j| = (g_j^{max} - g_j^{min})/\epsilon$. For simplicity, we assume ϵ evenly divides the range of each dimension ($g_j^{max} - g_j^{min}$). This ensures that, for a query point in a cell, the search for neighboring points within ϵ will be constrained to the adjacent cells, bounding the search and regularizing the instruction flow.

If all cells were indexed, this would yield $\prod_{j=1}^{n} |g_j|$ grid cells. For even moderate dimensionality datasets, the total number of cells would make the space complexity intractable. Thus, in contrast to [29] we elect to only store non-empty grid cells in our grid index, denoted as G. The space occupied by the index is a function of the data density and distribution and not the total (hyper)volume defined by $\prod_{j=1}^{n} g_j^{max} - g_j^{min}$.

C. Index Components

In this section, we describe the components of the index shown in Figure 2 (a) and in the following section describe the index search using an example grid in Figure 2 (b). To minimize the space occupied by the index structure, we use several supporting components, enabling us to only store non-empty grid cells. When finding the adjacent grid cells of a point, we determine if a grid-cell exists based on the

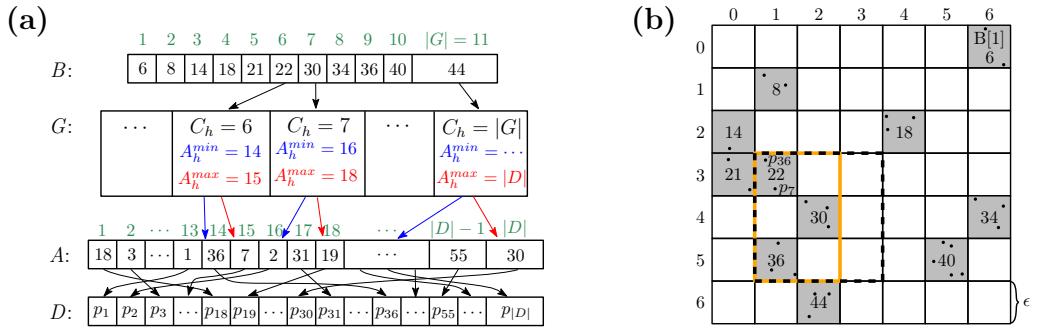


Fig. 2: Example of our point indexing strategy. (a) contains the indexing structure we employ and (b) illustrates the resulting grid of points in 2-D space.

coordinates of a point and a grid cell lookup array, denoted as B. The number of non-empty grid cells is denoted as $|G|$, thus $|G| = |B|$; however, both sizes are dependent on the data distribution of D. B simply maps a cell's linearized coordinate computed from the cell's n-dimensional coordinates to its location in G. To store the points in grid cells, we utilize a lookup array denoted as A. Each non-empty grid cell, denoted as C_h, $h \in \{1, \ldots, |G|\}$, is stored as a linearized cell id, and contains the minimum and maximum ranges in A that map to D. Thus, C_h stores a range $[A_h^{min}, A_h^{max}]$. Since A provides a direct mapping between D and the points inside each grid cell, $|A| = |D|$. A masking array is also computed to reduce the search space by filtering the cells as a function of the n-dimensional coordinates. We store only the coordinates of grid cells that are non-empty in each dimension. We denote this masking array as M_j, where $j \in \{1, \ldots, n\}$.

D. Index Search

We illustrate searching the index. Figure 2 (b) shows a 2-D example grid, where non-empty cells are shaded and are labeled in lexicographic order. In the figure, there are 11 non-empty cells, thus $|B| = |G| = 11$. Cell $C_h = 7$ (linear id 30) contains a point $a \in D$, which requires calculating the neighbors within ϵ of the adjacent grid cells. We begin by storing the range in each dimension of the neighboring grid cells of $C_h = 7$, in a range for each dimension, denoted as O_j, where $j = 1, \ldots, n$, which yields $O_1 = [1, 3]$ and $O_2 = [3, 5]$ in the respective dimensions (cells outlined with the black dashed lines in Figure 2 (b)). We then compare O_j to the masking array M_j. In the example, $M_1 = [1, 2]$ (in $j = 1$, there are no points in the 3rd cell column), and $M_2 = [3, 5]$. We then take $O_j \cap M_j$, which yields $[1, 2]$, and $[3, 5]$, respectively (cells outlined in orange in Figure 2 (b)). Then, with the ranges in each dimension, for each cell (6 in total), we compute the linear coordinate and binary search B (Figure 2 (a)) to see if the linear coordinate exists. If so, we compute the distances to the points in the cell and filter the points with distances that are $> \epsilon$. For example, C_6 has linear coordinate 22, and the points in D are found using A, as $\{A[A_h^{min}], \ldots, A[A_h^{max}]\}$ which yields $\{p_{36}, p_7\}$.

As an alternative to the lookup array A, one could store the points within each cell, C_h, thus mapping G directly to D. However, this would require allocating a constant amount of memory per cell, the size of which would be the cell containing the greatest number of points. This would waste a significant amount of memory. Likewise, we could store all cells, including those that are empty, but that would increase $|G|$ exponentially with the number of dimensions, potentially causing the index to require more memory than the global memory capacity. With our strategy, the space complexity required by the index is $O(|B| + |G| + |A|)$, where $|B| = |G|$ and $|A| = |D|$. Since our cells contain at least one point, our space complexity simplifies to $O(|D|)$.

The curse of dimensionality [11] is expected to reduce the pruning efficiency of index searches [14] in higher dimensions. The number of adjacent cells to a query cell is 3^n, and this leads to a significant number of cells to check to see if they exist in B. However, since average density decreases in higher dimensions, fewer adjacent cells to a query cell will be non-empty, allowing us to ignore many neighbor cells.

E. Global Memory GPU Kernel

In this section we provide details of GPUSELFJOIN-GLOBAL, the main GPU kernel of our algorithm, with pseudocode provided in Algorithm 1. We refer to Figure 2 when illustrating the kernel, and thus the kernel is presented in 2-D. For this kernel, we employ $|D|$ threads, where each thread considers a single point and finds all neighbors within ϵ using the grid-based index. Threads do not utilize shared memory in this kernel. The GPU kernel takes as input D, A, G, B, M, and, ϵ. Each thread begins by getting its global thread id and checking to see if the global id is larger than D (lines 2–3). The thread then stores its point in registers (line 5) and computes index ranges of adjacent cells in each dimension (line 6), corresponding to the black dashed box in Figure 2 (b). The ranges are then filtered in each dimension using M (line 7), resulting in the orange box in Figure 2 (b). The thread then iterates over these filtered ranges in each dimension (lines 8-9) to create 2-D coordinates that are used to compute a linear coordinate for each non-empty neighbor cell (line 10). In the example in Figure 2, the two loops generate the linearized ids: 22, 23, 29, 30, 36, 37 (orange outline). Each of these are searched in B, resulting in the non-empty cells 22, 30, and 36.

For each of these cells, the thread finds the points contained therein using array G to get the minimum and maximum ranges in A (lines 12–13). The thread then iterates over all points in this range (line 14), computing the distance from the query point (lines 15–16). If the distance computed is within ϵ, then the result is stored as a key/value pair (line 17), where the key is the query point id and the value is the point found to be within the ϵ distance. After the kernel's execution, we sort the key/value pairs, and transfer the result to the host.

Our example in Algorithm 1 illustrates GPUSELFJOINGLOBAL in 2-D (i.e., $n = 2$). When $n > 2$, additional loops are required after the loops on lines 8–9, and the adjacent cell ids and filtered cell id ranges (lines 6–7) also compute these ranges for the additional dimensions.

Algorithm 1 The GPUSELFJOINGLOBAL Kernel.

```
1:  procedure GPUSELFJOINGLOBAL(D, A, G, B, M, ε)
2:      gid ← getGlobalId()
3:      if gid ≥ |D| then return
4:      resultSet ← ∅
5:      point ← D[gid]
6:      adjRangesArr[n] ← getAdjCells(point)
7:      filteredRngs[n] ← maskCellRange(M,adjRangesArr[n])
8:      for dim1 ∈ filteredRngs[1].min, ..., filteredRngs[1].max do
9:          for dim2 ∈ filteredRngs[2].min, ..., filteredRngs[2].max do
10:             linearID ← getLinearCoord(dim1,dim2)
11:             if linearID ∈ B then
12:                 lookupMin ← A[G[linearID].min]
13:                 lookupMax ← A[G[linearID].max]
14:                 for candidateID ∈ {lookupMin,...,lookupMax} do
15:                     result ← calcDistance(point, D[candidateID])
16:                     if result ≤ ε then
17:                         atomic: resultSet ← resultSet ∪ result
    return
```

V. OPTIMIZATIONS

The indexing scheme presented in Section IV provides a general method for us to efficiently solve the self-join problem for datasets ranging from 2 to 6 dimensions. In this section, we discuss optimizations that further improve the performance of our self-join algorithm on the GPU. Since this work focuses on datasets from 2–6 dimensions, we focus on optimizations that address the challenges of processing low dimensional data.

A. Batching the Result Set

Batching schemes are required by many GPU-accelerated approaches that process large volumes of data. In low dimensionality, the hyper-volume of the space is small in comparison to high dimensionality, so more points co-occur in the same region, leading to larger result set sizes (e.g., Figure 1). Since there are potentially a large number of points within the ϵ distance, a bottleneck is the data transfer of the result set between the GPU and host. We leverage the work of [29] that calculated the ϵ-neighborhood of 2-D points on the GPU for use in the DBSCAN clustering algorithm [1]. In our context, without incrementally batching the result set it may not be possible to process low-dimensional data in a single kernel invocation as the result set may exceed global memory capacity. Furthermore, the batching scheme allows us to overlap GPU computation with bidirectional data transfers between host and GPU. Thus, even for workloads that would not overflow the GPU's global memory capacity, it is more efficient to use the batching scheme. In all experiments, the minimum number of batches is set to 3.

B. Avoiding Duplicate Calculations

Euclidean distance is reflexive, so if $p \in D$ is within ϵ-distance of $q \in D$, then $q \in D$ is within ϵ of $p \in D$. Thus, if we can evaluate pairs of neighboring points only *once*, we can report both corresponding ordered pairs and eliminate redundant work. We present a general method of accomplishing this for n dimensions, which we call "uni-directional comparison" (UNICOMP).

We consider each dimension separately and, for each point contained in a cell that has an *odd* coordinate for the given dimension, we evaluate points in a specific set of neighbor cells. Figure 3 illustrates which neighbor cells we want to evaluate when the query point is located in a (blue) cell with an odd index in each of the first 3 dimensions.

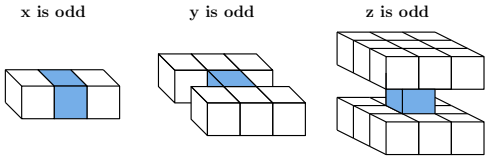


Fig. 3: Illustration of the UNICOMP optimization. The blue cell is our source cell and the white cells drawn around it are those we want to evaluate if the source cell index at the particular dimension is odd.

In the example of $n = 3$ (x, y, and z coordinates), we first consider the x-index. If cell C_a containing the source point has an *odd* x-index, we evaluate the neighbor cells that differ by x-index but share the same y and z indices as C_a. If the cell has an even x-index, we do nothing. We then consider the y-index: if odd, we evaluate all neighbor cells that differ by y but share the z-index as C_a. Finally, we consider z; if odd, we evaluate all neighbor cells with z-index that differs from C_a. Pseudocode illustrating this process for 3 dimensions (x, y, and z) is presented in Algorithm 2.

We apply this pattern when considering which neighbor cells to evaluate for each point p. Whenever a point q is found within the ϵ distance of p, we add both (p, q) and (q, p) to our result set. UNICOMP reduces both the index search overhead (cell evaluations) and Euclidean distance calculations roughly by a factor of two. Thus, we expect the UNICOMP optimization to improve overall performance by this factor.

VI. EXPERIMENTAL EVALUATION

A. Datasets

We utilize both real and synthetic datasets to evaluate the performance of our approaches. Our synthetic datasets (*Syn*) assume data is uniformly distributed and independent in each dimension. As will be elaborated on in the experimental evaluation, uniformly distributed data represents a worst-case scenario for our GPU grid indexing scheme because it

Algorithm 2 The UNICOMP access pattern 3 dimensions

```
1:  procedure UNICOMP 3D(point, C_a, filteredRngs, B)
2:      if C_a.x is odd then
3:          for x ∈ filteredRngs[1] do
4:              if x ≠ C_a.x then
5:                  linearID ← getLinearCoord(x, C_a.y, C_a.z)
6:                  if linearID ∈ B then
7:                      ComparePoints(point, linearID)
8:      if C_a.y is odd then
9:          for x ∈ filteredRngs[1] do
10:             for y ∈ filteredRngs[2] do
11:                 if y ≠ C_a.y then
12:                     linearID ← getLinearCoord(x, y, C_a.z)
13:                     if linearID ∈ B then
14:                         ComparePoints(point, linearID)
15:     if C_a.y is odd then
16:         for x ∈ filteredRngs[1] do
17:             for y ∈ filteredRngs[2] do
18:                 for z ∈ filteredRngs[3] do
19:                     if z ≠ C_a.z then
20:                         linearID ← getLinearCoord(x, y, z)
21:                         if linearID ∈ B then
22:                             ComparePoints(point, linearID)
    return
```

maximizes the number of non-empty cells, leading to higher search overhead in comparison to datasets with regions of higher density, relative to the average data density. We generate datasets in multiple dimensions with 2 and 10 million points to observe how performance varies as a function of data density and dimensionality. Each data point is represented as a 64-bit floating point value uniformly distributed in the range [0,100] in each dimension. Additionally, we evaluate performance using the following real-world datasets. The *SW-* dataset contains the latitude/longitude and the total electron content in the ionosphere [31]. We use this dataset in either 2 or 3 dimensions: in 2-D, we use the coordinates of the point, and in 3-D we include the total electron content value. The *SDSS-* datasets are galaxies from the Sloan Digital Sky Survey, data release 12 [32]. Galaxies are represented in 2-D, spanning redshift (z) of $0.30 \leq z \leq 0.35$. A summary of dataset properties are outlined in Table I.

TABLE I: Dataset, data points, $|D|$, and dimension, n.

Dataset	$\|D\|$	n	Dataset	$\|D\|$	n
Syn-					
Syn2D2M	2×10^6	2	*Syn2D10M*	10×10^6	2
Syn3D2M	2×10^6	3	*Syn3D10M*	10×10^6	3
Syn4D2M	2×10^6	4	*Syn4D10M*	10×10^6	4
Syn5D2M	2×10^6	5	*Syn5D10M*	10×10^6	5
Syn6D2M	2×10^6	6	*Syn6D10M*	10×10^6	6
Real World: *SW-*, *SDSS-*					
SW2DA	1,864,620	2	*SW2DB*	5,159,737	2
SW3DA	1,864,620	3	*SW3DB*	5,159,737	3
SDSS2DA	2×10^6	2	*SDSS2DB*	15,228,633	2

B. Experimental Methodology

The GPU code is written in CUDA [33] and executed on an nVIDIA TITAN X (Pascal architecture) GPU with 12 GiB of global memory. All C/C++ host code is compiled with the GNU compiler with the O3 optimization flag. The platform has $2 \times 16 = 32$ Intel E5-2683 v4 2.1 GHz CPU cores. The self-join CUDA kernel (Algorithm 1) is configured to run with 256 threads per block, and uses 64-bit double precision floats. All results are averaged over 3 trials. We denote our GPU approach as GPU-SJ.

Reference Implementation (CPU-RTREE): We compare the performance of GPU-SJ to a sequential reference implementation using an R-tree [9] index, denoted as CPU-RTREE. The performance of the R-tree search is sensitive to the insertion order of the data, and co-located data should be inserted together (e.g., using a Hilbert curve [34]). Thus, in all experiments, we first sort the data into bins of unit length in each dimension. This ensures that internal nodes of the R-tree do not encompass too much empty space. Since we focus on index search performance, we omit index construction time. We note, however, that inserting points into the grid in GPU-SJ requires far less work than constructing the R-tree.

As discussed in Section I, the search-and-refine strategy that is reflected in CPU-RTREE is efficient in low dimensionality; however, index search performance degrades in higher dimensions. For this reason, *we only focus on dimensions 2–6*.

State-of-the-art (SUPEREGO): The Super-EGO algorithm [8] for the CPU performs fast self-joins on multi-dimensional data. It has been shown to outperform many other join algorithms on both low and high dimensional data and is considered state-of-the-art. As such, we compare our approach to the multi-threaded version of Super-EGO [8], using 32 threads on our 32 core platform. We execute the algorithm using 32-bit floats (execution with 64-bit floats failed). However, our GPU implementation uses 64-bit floats; therefore, the SUPEREGO algorithm is expected to be slower if it were executed with 64-bit floats (e.g., more cache misses would occur). Furthermore, SUPEREGO normalizes all data in the range [0,1] in each dimension. We modified our datasets accordingly, but in our figures, we show the non-normalized value of ϵ so that we can compare results. We validated consistency between our implementations by comparing the total number of neighbors within ϵ. In all time measurements, we use the total time to ego-sort and join. We are grateful to D. Kalashnikov for making his code publicly available.

GPU Brute Force Implementation: As dimensionality increases, there is increased index search overhead in any self-join operations that use an index. Thus, at some dimension, a brute force nested loop join $O(|D|^2)$ algorithm [8] that compares all points to each other is expected to be more efficient than using an index. Since this brute force approach compares all pairs of points, it is independent of ϵ. We also compare to a parallel brute force implementation on the GPU. The kernel simply uses $|D|$ threads, where each thread is assigned one point to compare to each of the other points in the dataset. We exclude the time to transfer the result set back to the host, and thus only execute a single kernel invocation. This represents a lower bound on the brute force implementation. Since the performance of the brute force algorithm is not dependent on ϵ, we only run the brute force algorithm for a single value of ϵ on a given dataset. This GPU brute force

implementation demonstrates that the performance gains of our approaches are *not* due to the high throughput of the GPU.

C. Results

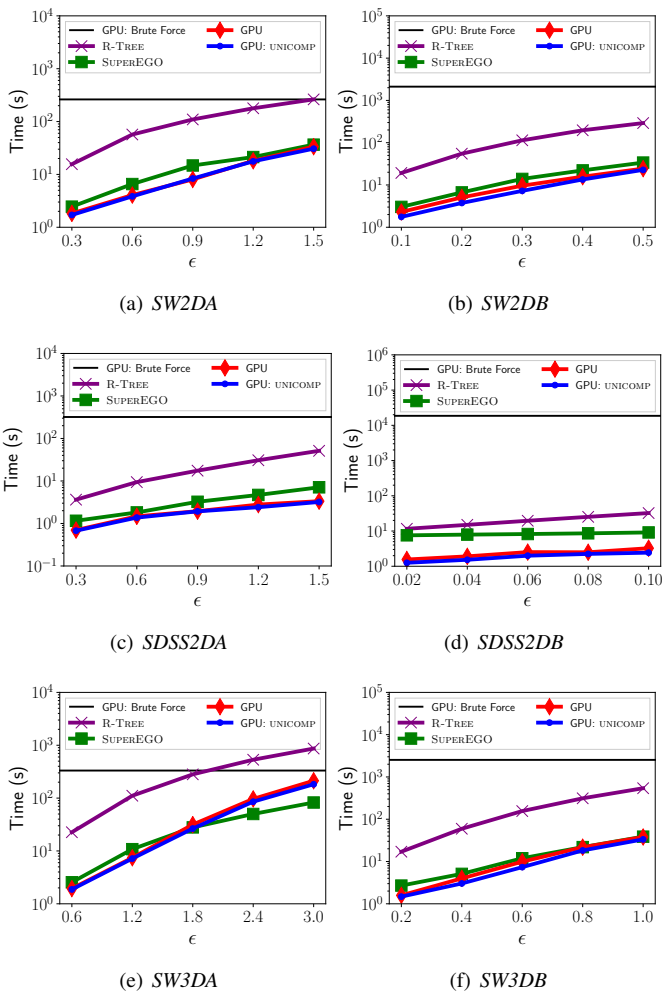
As the search distance, ϵ, increases, the performance of the self-join degrades for two reasons: (i) more comparisons are required to determine if two points are within ϵ of each other; and, (ii) the performance of index searches degrade as the search volume increases. Our grid-based index is less susceptible to the effect in (ii) above because the search is limited to adjacent grid cells of a given query point.

Real-world datasets: Figure 4 plots the response time vs. ϵ for the real-world datasets, which span 2–3 dimensions. The figure is plotted on log scale so that we can observe differences in response times across multiple orders of magnitude (we summarize speedup in a subsequent figure). Across all datasets, GPU-SJ outperforms CPU-RTREE. Furthermore, GPU-SJ outperforms the state-of-the-art algorithm, SUPEREGO (executed in parallel with 32 threads) across most scenarios. We find that on *SW3DA* when $\epsilon > 1.8$, SUPEREGO outperforms GPU-SJ with UNICOMP. However, on the largest dataset, *SDSS2DB* (containing 15 million points), GPU-SJ with UNICOMP achieves up to $\sim$6$\times$ speedup over SUPEREGO. Although GPU-SJ is faster than the brute force method for all of our experiments, on the *SW3DA* dataset (Figure 4 (e)), brute force outperforms CPU-RTREE at $\epsilon > 1.8$. This is because the ϵ values are quite large in this experiment, increasing the number of distance calculations and diminishing the efficacy of the index.

Synthetic datasets: Figures 5 and 6 show the response time vs. ϵ for synthetic datasets with 2 and 10 million datapoints, respectively, spanning 2–6 dimensions. While the results on the 2-D datasets are consistent with the observations for the real-world datasets in Figure 4, the results for higher dimensions are not. The performance of UNICOMP improves with dimensionality, particularly when the number of dimensions, $n \geq 3$ (comparing Figures 5 and 6 (b)-(e)). Across all synthetic datasets, GPU-SJ (with UNICOMP) outperforms CPU-RTREE. Furthermore, GPU-SJ with UNICOMP outperforms SUPEREGO on nearly all experiments.

Impact of data distribution on performance: We use uniformly distributed synthetic datasets because: (i) the performance of searches is data-dependent and uniformly distributed data is an average case compared to sparse datasets or datasets with over-dense regions; and (ii) uniformly distributed data is a worst-case scenario for the GPU-SJ grid index. Datasets having more over-dense regions will have fewer non-empty cells, resulting in fewer cells to search. In contrast, uniformly distributed data will have more cells with fewer points contained within, maximizing the number of non-empty cells. Thus, uniformly distributed data has greater *search* overhead than data distributions with highly varying densities (i.e., real-world data).

SUPEREGO also tends to perform better on non-uniform data for many of the same reasons as GPU-SJ. However, SUPEREGO reorders the dimensions to improve cell pruning as a function of data distribution [8]. We find that across the uniformly distributed synthetic datasets, the performance of SUPEREGO degrades with ϵ in a similar manner to GPU-SJ.



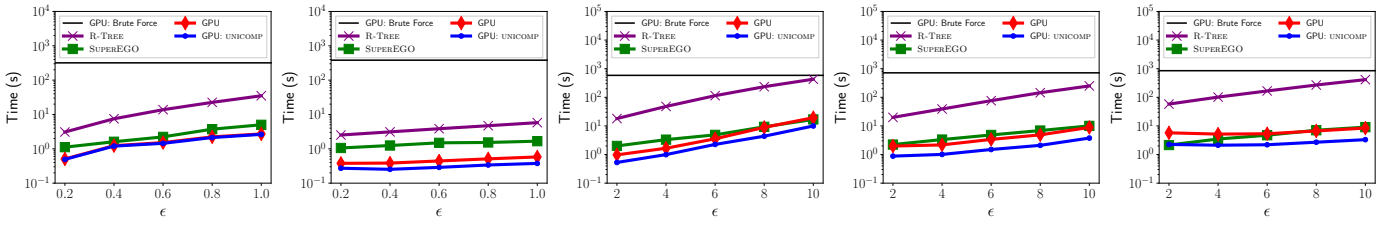
(a) *SW2DA* (b) *SW2DB*

(c) *SDSS2DA* (d) *SDSS2DB*

(e) *SW3DA* (f) *SW3DB*

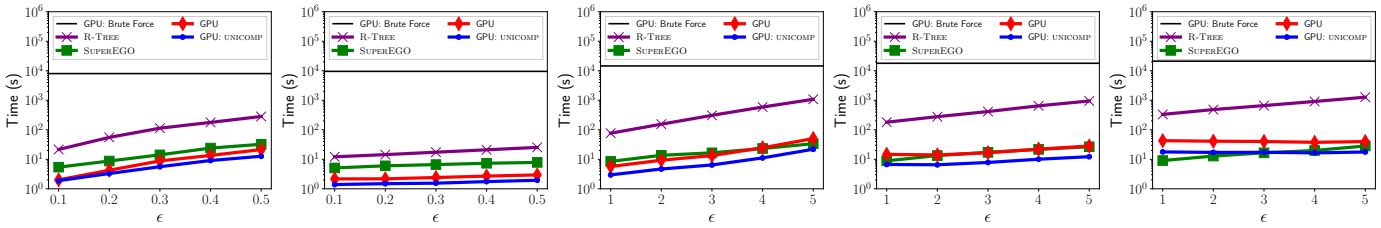
Fig. 4: Response time vs. ϵ on the real-world datasets, *SW-* (a, b, e, f) and *SDSS-* (c, d). The *SDSS-* datasets are in 2-D and *SW-* span 2–3 dimensions.

GPU-SJ vs. CPU-RTREE: Figure 7 plots the speedup of GPU-SJ with UNICOMP over CPU-RTREE vs. ϵ for each dataset from Figures 4, 5, and 6. The lowest performance gain over CPU-RTREE occurs on *SDSS2DA* and *SW2DA* (i.e., the smallest workloads). When $2 \leq n \leq 3$, we see fairly consistent speedups, regardless of ϵ or $|D|$. This indicates that the speedup of GPU-SJ over CPU-RTREE is less a function of data distribution (e.g., uniform vs. skewed). Rather, data dimensionality dictates the rate at which ϵ degrades index performance, and this relationship differs between GPU-SJ and CPU-RTREE. This suggests that the speedup we see with synthetic datasets when $4 \leq n \leq 6$ (Figure 7) will be consistent with real-world datasets of the same dimensionality. Furthermore, index degradation with dimensionality explains why the performance gains are largest on the higher dimen-



(a) *Syn2D2M* (b) *Syn3D2M* (c) *Syn4D2M* (d) *Syn5D2M* (e) *Syn6D2M*

Fig. 5: Response time vs. ϵ on the 2–6 dimensional synthetic datasets with 2×10^6 points.



(a) *Syn2D10M* (b) *Syn3D10M* (c) *Syn4D10M* (d) *Syn5D10M* (e) *Syn6D10M*

Fig. 6: Response time vs. ϵ on the 2–6 dimensional synthetic datasets with 10^7 points.

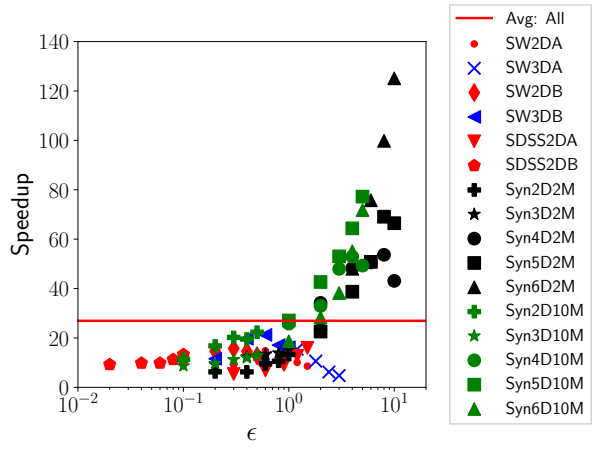


Fig. 7: Speedup of GPU-SJ with UNICOMP over CPU-RTREE (1 thread) for all real-world and synthetic datasets plotted on log scale to capture orders of magnitude differences in ϵ. Figure derived from Figures 4, 5, and 6. The red line shows the average speedup (26.9×).

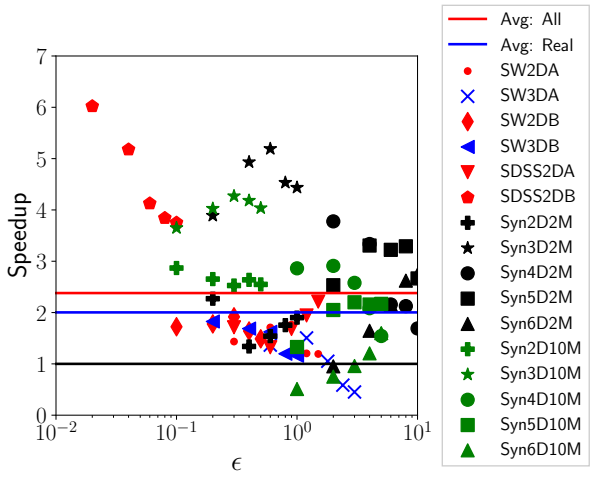
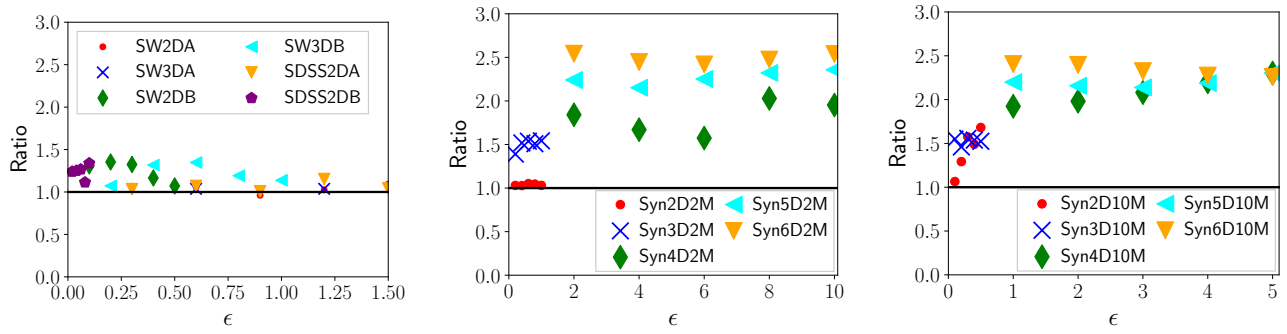


Fig. 8: Speedup of GPU-SJ with UNICOMP over SUPEREGO (32 threads) for all real-world and synthetic datasets plotted on log scale to capture orders of magnitude differences in ϵ. Figure derived from Figures 4, 5, and 6. The black line shows where our approach achieves a speedup (or slowdown). The red and blue lines show the average speedup across all of the datasets and real-world datasets, respectively.

sional datasets (up to 125×). Across all datasets, GPU-SJ is on average 26.9× faster than CPU-RTREE.

GPU-SJ vs. SUPEREGO: Figure 8 plots the speedup of GPU-SJ with UNICOMP over SUPEREGO (with 32 threads). Similar to Figure 7, the GPU performance gain is lowest on scenarios with small workloads. We find that there are only 6 instances where SUPEREGO outperforms GPU-SJ (below the black line). The average speedup on real-world datasets is $\sim 2\times$ (blue line), while the average across all datasets is 2.38× (red line). On average GPU-SJ with UNICOMP is 138% faster than SUPEREGO. As expected, SUPEREGO performs worse on the synthetic datasets, as it cannot benefit from dimensionality reordering on uniformly distributed data.

Many-core GPUs can be leveraged in heterogeneous environments for data-intensive (self-) join operations. SUPEREGO is a state-of-the-art join algorithm, and across nearly all scenarios, our GPU-SJ algorithm outperforms SUPEREGO



(a) Real: *SW-* ($2 \leq n \leq 3$), *SDSS-* ($n = 2$). (b) *Syn-* $2 \leq n \leq 6$, $|D| = 2 \times 10^6$. (c) *Syn-* $2 \leq n \leq 6$, $|D| = 10^7$.

Fig. 9: Impact of the UNICOMP optimization shown as the ratio of the response times of the GPU self-join without and with UNICOMP. Panels (a), (b), and (c), are derived from Figures 4, 5, and 6, respectively. Points above the horizontal line indicate that UNICOMP leads to a performance gain, whereas points below indicate a slowdown due to associated overheads.

TABLE II: Selected kernel metrics of GPU-SJ without and with UNICOMP. All ratios given in the quantity of the metric with UNICOMP divided by the value without the UNICOMP optimization.

Dataset	ϵ	Ratio Resp. Time (Figure 9)	Theoretical Occupancy GPU	Unified Cache Bandwidth Utilization (GB/s) GPU	Theoretical Occupancy (GPU: UNICOMP)	Unified Cache Bandwidth Utilization (GB/s) (GPU: UNICOMP)	Ratio Occupancy	Ratio Cache Utilization
SW2DA	0.3	1.07	100%	597.87	75%	460.96	0.75	0.77
SDSS2DA	0.3	1.03	100%	1157.07	75%	864.26	0.75	0.75
Syn5D2M	8	**2.32**	62.5%	306.45	50%	578.14	0.8	**1.88**
Syn6D2M	8	**2.47**	62.5%	293.31	50%	469.24	0.8	**1.59**

as executed on 32 CPU cores. Also, recall that our results for SUPEREGO are using 32-bit data points, while GPU-SJ uses 64-bit on all scenarios; therefore, we expect further performance gains over SUPEREGO if 64-bit floats are used.

Performance characterization of UNICOMP: As previously mentioned, while UNICOMP decreases the number of point comparisons and cells searched by a factor of ∼2, it does not yield a corresponding decrease in response time. This leads to unexpected speedups over CPU-RTREE and SUPEREGO as a function of ϵ and dataset size, as mentioned above. Figure 9 shows the ratio of the response times of GPU-SJ with and without UNICOMP, derived from the results shown in Figures 4, 5, and 6 (by dividing the respective response times). While there are are a few scenarios in Figure 9 (a) for which UNICOMP results in a slight performance loss due to overhead, the resulting slowdowns are negligible; thus, UNICOMP can be used without concern of significant performance degradation.

When using UNICOMP on real-world datasets (Figure 9 (a)), the response time ratios are within 1.5×, and not 2× as expected. However, the higher dimensionality ($n \geq 3$) results shown in Figure 9 (b) and (c) demonstrate that UNICOMP achieves some performance gains that are $\geq$ 2×. This is a surprising result, as UNICOMP reduces the number of cells and points searched by a factor of ∼2. To understand this phenomenon, we consider two cases: when the response time ratio is < 2 and when it is > 2. We use the nVIDIA Visual Profiler [35] to collect two metrics during execution: *occupancy* and *unified cache utilization*. Occupancy is a measure of the number of threads running simultaneously on the GPU. Thus, low occupancy can result in under-utilization of GPU cores. On the nVIDIA Maxwell and Pascal GPUs, the unified (L1) cache is a coalescing buffer for memory accesses [36]. Particularly relevant is the caching of global loads. We compare the occupancy and unified L1 cache bandwidth utilization on the self-join kernels (without and with UNICOMP).

Table II shows two datasets with response time ratios of GPU-SJ (without/with UNICOMP) < 2 (*SW2DA*, *SDSS2DA*) and > 2 (*Syn5D2M*, *Syn6D2M*). In all datasets, using UNICOMP results in lower occupancy due to more registers being used per thread. As expected higher dimensionality also reduces occupancy due to register usage. Thus, while UNICOMP reduces occupancy, we notice that the relative cache utilization depends on the dataset. For those with response time ratios < 2 (*SW2DA*, *SDSS2DA*), UNICOMP reduces cache utilization. However, when ratios are > 2 (*Syn5D2M*, *Syn6D2M*), UNICOMP *increases* cache utilization. Thus, we attribute the increased performance of UNICOMP on higher dimensional datasets to a higher degree of temporal locality in the L1 cache. This explains the variance in performance from UNICOMP, despite it decreasing the work by a factor of ∼2.

VII. DISCUSSION & CONCLUSION

The self-join is a widely used operation in many data intensive search algorithms and Big Data applications. We demonstrate that our algorithm, GPU-SJ, that combines a

grid-based index that is suited for the GPU with batched result reporting and duplicate search removal (UNICOMP), outperforms the multi-threaded state-of-the-art SUPEREGO algorithm with an average speedup of 2.38×. GPU-SJ also significantly outperforms the search-and-refine strategy, CPU-RTREE.

Due to the large range of relevant related work (Section II), we summarize the novelty of our techniques as follows:
•In low-dimensionality, there are likely to be many neighbors within the query distance ϵ. We batch the results to process self-join result sets that exceed the GPU's global memory capacity. The batching scheme overlaps computation and communication to amortize the host-GPU overheads.
•We motivate the use of a grid index for the GPU due to its bounded search, and thus greater regular instruction flow in comparison to index-trees. While grids have been used on the CPU for the similarity join [8], [21], we have not found any works that apply a grid on the GPU to compute the self-join.
•By virtue of utilizing a grid, our novel UNICOMP optimization reduces the number of grid cells searched and distance calculations, is generalizable to n-dimensions, and improves cache utilization when $n > 3$.

Future work includes applying this work to other spatial searches, such as kNN, and determining if this work is relevant for self-joins on higher dimensional datasets.

ACKNOWLEDGMENT

We thank Frédéric Loulergue, and UHHPC at the University of Hawaii for the use of their platforms.

REFERENCES

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

[2] K. Koperski and J. Han, "Discovery of spatial association rules in geographic information databases," in *Advances in spatial databases*. Springer, 1995, pp. 47–66.

[3] R. J. Bayardo, Y. Ma, and R. Srikant, "Scaling up all pairs similarity search," in *Proc. of the Intl. Conf. on World Wide Web*, 2007, pp. 131–140.

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

[5] R. Agrawal, C. Faloutsos, and A. Swami, "Efficient similarity search in sequence databases," *Foundations of data organization and algorithms*, pp. 69–84, 1993.

[6] C. Böhm, B. Braunmüller, M. Breunig, and H.-P. Kriegel, "High performance clustering based on the similarity join," in *Proc. of the Intl. Conf. on Information and Knowledge Management*, 2000, pp. 298–305.

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

[8] D. V. Kalashnikov, "Super-ego: fast multi-dimensional similarity join," *The VLDB Journal*, vol. 22, no. 4, pp. 561–585, 2013.

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

[10] E. H. Jacox and H. Samet, "Metric space similarity joins," *ACM Trans. Database Syst.*, vol. 33, no. 2, pp. 7:1–7:38, 2008.

[11] R. E. Bellman, *Adaptive control processes: a guided tour*. Princeton university press, 1961.

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

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

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

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

[16] A. Arasu, V. Ganti, and R. Kaushik, "Efficient exact set-similarity joins," in *Proc. of the Intl. Conf. on Very Large Data Bases*, 2006, pp. 918–929.

[17] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger, "The r*-tree: An efficient and robust access method for points and rectangles," in *Proc. of the ACM Intl. Conf. on Management of Data*, 1990, pp. 322–331.

[18] S. Berchtold, D. A. Keim, and H.-P. Kriegel, "The x-tree: An index structure for high-dimensional data," in *Proc. of the 22th Intl. Conf. on Very Large Data Bases*, 1996, pp. 28–39.

[19] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Communications of the ACM*, vol. 18, no. 9, pp. 509–517, 1975.

[20] R. A. Finkel and J. L. Bentley, "Quad trees a data structure for retrieval on composite keys," *Acta informatica*, vol. 4, no. 1, pp. 1–9, 1974.

[21] C. Böhm, B. Braunmüller, F. Krebs, and H.-P. Kriegel, "Epsilon grid order: An algorithm for the similarity join on massive high-dimensional data," in *ACM SIGMOD Record*, vol. 30, no. 2, 2001, pp. 379–388.

[22] K. Yang, B. He, R. Fang, M. Lu, N. Govindaraju, Q. Luo, P. Sander, and J. Shi, "In-memory grid files on graphics processors," in *Proc. of the 3rd Intl. Workshop on Data Management on New Hardware*, 2007, pp. 5:1–5:7.

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

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

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

[26] J. Kim and B. Nam, "Co-processing heterogeneous parallel index for multi-dimensional datasets," *Journal of Parallel and Distributed Computing*, vol. 113, pp. 195 – 203, 2018.

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

[28] M. D. Lieberman, J. Sankaranarayanan, and H. Samet, "A fast similarity join algorithm using graphics processing units," in *IEEE 24th Intl. Conf. on Data Engineering*, 2008, pp. 1111–1120.

[29] M. Gowanlock, C. M. Rude, D. M. Blair, J. D. Li, and V. Pankratius, "Clustering Throughput Optimization on the GPU," in *Proc. of the IEEE Intl. Parallel and Distributed Processing Symposium*, 2017, pp. 832–841.

[30] M. Burtscher, R. Nasre, and K. Pingali, "A quantitative study of irregular programs on GPUs," in *IEEE Intl. Symposium on Workload Characterization*, 2012, pp. 141–151.

[31] ftp://gemini.haystack.mit.edu/pub/informatics/dbscandat.zip, accessed 17-September-2017.

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

[33] NVIDIA, "CUDA programming guide 9.0," 2017. [Online]. Available: http://docs.nvidia.com/cuda

[34] I. Kamel and C. Faloutsos, "On Packing R-trees," in *Proc. of the Second Intl. Conf. on Information and Knowledge Management*, ser. CIKM '93, 1993, pp. 490–499.

[35] NVIDIA, "Nsight," 2015. [Online]. Available: http://www.nvidia.com/object/nsight.html

[36] http://docs.nvidia.com/cuda/pascal-tuning-guide/index.html#memory-throughput, accessed 19-January-2018.