Accelerating the Similarity Self-Join Using the GPU

Michael Gowanlock[a,1,*], Ben Karsin[b,c,2]

[a]School of Informatics, Computing, and Cyber Systems, Northern Arizona University, Flagstaff, AZ, U.S.A., 86011
[b]Department of Information and Computer Sciences, University of Hawaii at Manoa, Honolulu, HI, U.S.A., 96822
[c]Department of Computer Sciences, Université libre de Bruxelles, Brussels, Belgium

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 fundamental building block for the field of databases and data mining. In low dimensionality, there are several challenges associated with efficiently computing the self-join on the graphics processing unit (GPU). Low dimensional data results in higher data densities, causing a significant number of distance calculations and a large result set, and as dimensionality increases, index searches become increasingly exhaustive. We propose several techniques to optimize the self-join using the GPU that include a GPU-efficient index that employs a bounded search, a batching scheme to accommodate large result sets, and duplicate search removal with low overhead. Furthermore, we propose a performance model that reveals bottlenecks related to the result set size and enables us to choose a batch size that mitigates two sources of performance degradation. Our approach outperforms the state-of-the-art on most scenarios.

Keywords:
GPGPU, In-memory database, Query optimization, Self-join

1. Introduction

The similarity self-join is used as a subroutine in many algorithms. The self-join is described as follows: given a dataset of objects, find all pairs of objects that share 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 in space. However, in principle, distance similarity is a common metric, and can be applied to non-spatial applications as well. In this paper, we focus on the distance similarity self-join that finds all pairs of points that are within a distance ϵ of each other, using Euclidean distance. In the field of databases, the self-join is a special case of a join operation between two tables (i.e., a table is joined on itself). Therefore, query optimizations for the self-join are largely applicable to the regular join operation.

Similarity self-joins are building blocks of existing data analysis algorithms. For example, the DBSCAN clustering algorithm relies on 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 data analytics 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 increasing dimensionality. We consider data up to 6-D, which is within the regime where index search performance degradation is not prohibitive for canonical indexing schemes, and thus allows us to directly compare our approach to the search-and-refine approach.

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 [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, many costly Euclidean distance calculations are still needed.

*Corresponding author. Michael Gowanlock, School of Informatics, Computing, and Cyber Systems (Building #90) 1295 S. Knoles Dr., Flagstaff, AZ, 86011, U.S.A.
[1]E-mail: michael.gowanlock@nau.edu
[2]E-mail: benjamin.karsin@ulb.ac.be

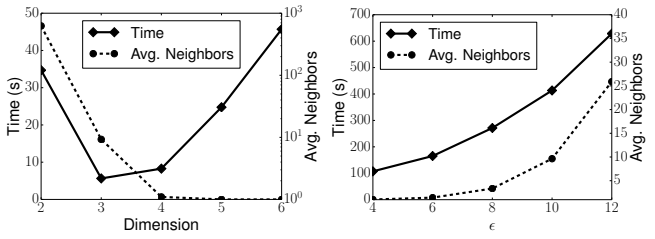


Figure 1: Problem overview: (a) Computing the distance similarity self-join on 2 million data points in 2–6 dimensions with the distance $\epsilon = 1$ using an R-tree index, where each dimension is of equal length and the points are uniformly distributed in a (hyper)cube. (b) Time vs. ϵ for the 6-D dataset shown in (a). Datasets (*Syn-*) are described in Section 5.1.

Second, while at 6-D there are very few neighbors within ϵ per point (Figure 1 (b) plots response time versus ϵ for 6-D data), the index search is more exhaustive. Thus, due to the well-known curse of dimensionality [11, 12, 13, 14], index performance degrades with increasing dimension, increasing self-join response time.

The modern Graphics Processing Unit (GPU) architecture is well-suited for the self-join problem. The self-join requires a significant number of independent index searches (one search per point in the database). The resulting candidate set must then be refined by performing distance calculations between points. For each point, these calculations are independent and can therefore be performed in parallel on the GPU. Since index searches are largely memory-bound, the high aggregate memory bandwidth on modern GPUs further improve performance. For example, the NVIDIA Volta architecture has an aggregate memory bandwidth of 900 GB/s [15], which is $\sim 10\times$ greater than the bandwidth between CPU and main memory. Thus, multi-core CPU architectures may suffer from more performance degradation due to memory bandwidth bottlenecks than GPUs. This is particularly the case in low-dimensional spaces where the distance calculation is less computationally expensive than in high dimensional feature spaces. Consequently, we propose a self-join algorithm that leverages the GPU to efficiently compute the distance similarity self-join. This article extends the preliminary work in [16] and makes the following contributions:

- We evaluate the performance of our GPU-accelerated self-join on 18 datasets (real-world and synthetic) that span 2–6 dimensions and are characterized by differing data distributions.
- We extend an efficient indexing strategy, enabling us to bound the search for nearby points and exploit the high memory bandwidth and parallelism afforded by the GPU.
- By carefully considering the domain decomposition provided by the index, we provide two selection strategies that eliminate duplicate searches and distance calculations. We show that one strategy results in less overhead, but the performance impact 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. The batching scheme enables overlapping data transfers between the host and GPU, and concurrent computation on the host and GPU.
- We demonstrate that uniformly distributed data constitutes a worst-case scenario for our grid-based indexing approach. We show that for a fixed dimension, our approach achieves similar performance gains on both real-world and uniformly distributed synthetic datasets.
- We show that the performance of our approach is dependent on the batch size that is processed at each kernel invocation. We develop a performance model that reveals a trade-off between two major sources of overhead. The performance model can be used to select a good batch size to optimize the performance of the GPU-accelerated self-join.

The paper is outlined as follows: Section 2 provides background information and formalizes the problem, Section 3 presents our GPU indexing strategy, Section 4 describes additional techniques used to improve self-join performance, Section 5 presents experimental results of our GPU-accelerated self-join algorithm and details our performance model, and Section 6 concludes the work.

2. Problem Statement and Background

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 pairs of points, $p_i, p_j \in D$, that are within the Euclidean distance, ϵ, of each other (the implementation of this is 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 $dist(a, b) = \sqrt{\sum_{j=1}^{n}(a(x_j) - b(x_j))^2}$. In database terminology, the self-join is denoted as $A \bowtie_\epsilon A$, where A is a dataset or table. It is clear that two different sets of points, A and B, can be implemented as joins in a very similar manner to any self-join algorithm, where $A \bowtie_\epsilon B$. Thus, the methods advanced in this work are applicable to joins on two different datasets as well. We consider the case where all processing occurs in-memory on the GPU and CPU.

There are 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 the GPU to improve performance in this work, we also consider related work that focuses on algorithmic transformations and GPU optimized indexes. Furthermore, performance may be dependent on the data distribution; therefore, we discuss possible effects of skewed datasets. We present an overview of each of these relevant areas below.

2.1. Search-and-refine

Several works have studied the similarity-join problem on the CPU [6, 17, 18, 3, 8]. 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 work utilizes the search-and-refine strategy and they evaluate their clustering approach using the R*-Tree [19] and X-tree [20] 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.

2.2. Employing Grids and Grid-based Indexes

Spatial indexes, such as the R*-Tree [19] 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 [21] or quad trees [22].

In contrast to these tree-based indexes, 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 tree-based indexes may be much faster at data retrieval on data distributions for which they are designed. Consequently, data-dependent tree-based indexes 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" [23] 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, the points with ϵ distance are limited to the adjacent grid cells of the cell containing the query point, and these cells can be pruned by determining how far away they lie based on the individual cell's n-dimensional coordinate.

Kalashnikov [8] advances an epsilon grid order [23] approach called SuperEGO that uses the data distribution to inform pruning the search, and is the current state-of-the-art for join operations. The SuperEGO approach uses a grid structure that is designed for multi-core CPU execution and uses the epsilon grid order non-materialized grid to prune the search space for relevant points. For instance, if a point in a given dimension is found in cell C_1, and a nearby point is found in cell C_3, where 1 and 3 are cells in the given dimension, then the points are separated by ϵ, and there is no possibility that these points are within ϵ of each other.

One of the key innovations of SuperEGO [8] is that it uses a dimensionality reordering strategy, which increases the discriminative power of the index to prune the search. This approach is useful in high dimensionality, where some of the dimensions may provide more discriminatory power than others. However, in this work we target low-dimensionality data, so the reordering strategy employed by SuperEGO may not provide significant performance benefits. The approach that we propose in this work focuses on GPU architectures, which have significantly different performance considerations, so, unlike SuperEGO, we use a *materialized* grid (in the form of a sparse grid structure). We assign each GPU thread a point, and all threads reuse the materialized grid to search all relevant cells for neighboring points that are within the ϵ distance. We also propose several optimizations, such as avoiding duplicate distance calculations, that can significantly improve the performance of our self-join on low-dimensional datasets. In summary, while SuperEGO employs a grid-based index structure, it differs significantly from the techniques presented in this work. In Section 5 we experimentally compare the performance of SuperEGO and our approach.

2.3. Indexes for the GPU

To reduce the number of distance calculations, several papers have advanced GPU-efficient indexes [24, 7, 25, 14, 26, 27, 28]. 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 [29]. To address this, Kim et al. [14] propose methods of partitioning R-tree structures to reduce branch divergence and increase parallelism. While they show that the *braided parallelism* approach achieves high query throughput, their approach focuses on using many GPU cores to reduce the latency of individual range queries. Thus, it is unclear if this approach is well-suited for very large batches of queries, such as the self-join that we focus on in this work. Unfortunately, we could not verify this experimentally. While we were able to obtain the source code from the authors, we were unable to successfully execute it on our platform.

The authors of [14] that developed the GPU R-tree described above later presented a hybrid CPU/GPU R-tree in [28]. A key insight from [28] 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) GPU index that suffers from additional overheads and lower parallel efficiency. This finding motivates our use of a non-tree-based indexing scheme that is well-suited to the GPU.

2.4. Data Distributions

Depending on the algorithm used to perform the self-join, performance can be dependent on data distribution. In particular, the performance of the self-join in high dimensionality can suffer from skewed data distributions, which is one motivation for the dimensionality reordering scheme of SuperEGO [8]. A hierarchical grid index for the GPU that is designed for efficient searches in skewed data distributions is presented in [24]. In our evaluation, we utilize both real-world and uniformly distributed data, where real-world data is often naturally skewed. As mentioned above, in high dimensions, index searches become increasingly exhaustive. Thus, to improve the pruning power of index searches, the data can be accessed based on the dimensions that improve the discriminatory power of the data in each dimension. However, the low-dimensional self-joins that we consider in this work are less susceptible to performance degradation due to skewed data because the searches are less exhaustive in low dimensionality than in higher dimensions. Thus, skewed data distributions are not expected to lead to significant performance degradation. Furthermore, and as will be discussed in our results, the performance of the self-join on uniformly distributed data may perform relatively worse than on skewed datasets.

2.5. GPU Similarity Joins

Bohm et al. [7] present an algorithm that solves the general similarity join problem by using 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, this approach on 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 [30] is another GPU algorithm that performs a similarity join on two datasets and targets high-dimensional data ($A \bowtie_\epsilon B$). The authors note that the GPU's data operations are limited, and they therefore compute the similarity join using sorting and searching primitives. The LSS algorithm sorts the first dataset using space-filling curves. Then, each point in the second dataset is queried to find its neighbors within ϵ on the first dataset. The search occurs in parallel using an interval query on one of the space filling curves in the first dataset. Our GPU-SJ algorithm uses a different approach than LSS: we use a grid-based indexing structure, which, as we demonstrate, allows us to avoid duplicate calculations. The target dimensionality of LSS is higher than that considered here, and so we do not compare our work to LSS.

3. GPU Indexing

3.1. 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 (e.g., 2-D [31]), it is intractable in higher dimensions. Thus, in contrast to previous work [31], we do not index empty cells.

Tree-based indexes 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 [29]. Also, the memory access patterns associated with such tree-traversals result in poor cache utilization. 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 a tree-based index. 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 [32].

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

3.2. Index Properties

Our index leverages the work of [31]. 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 prior work [31] we only store non-empty grid cells. As such, our space occupied by our grid index is a function of the data density and distribution and not the total bounding (hyper)volume.

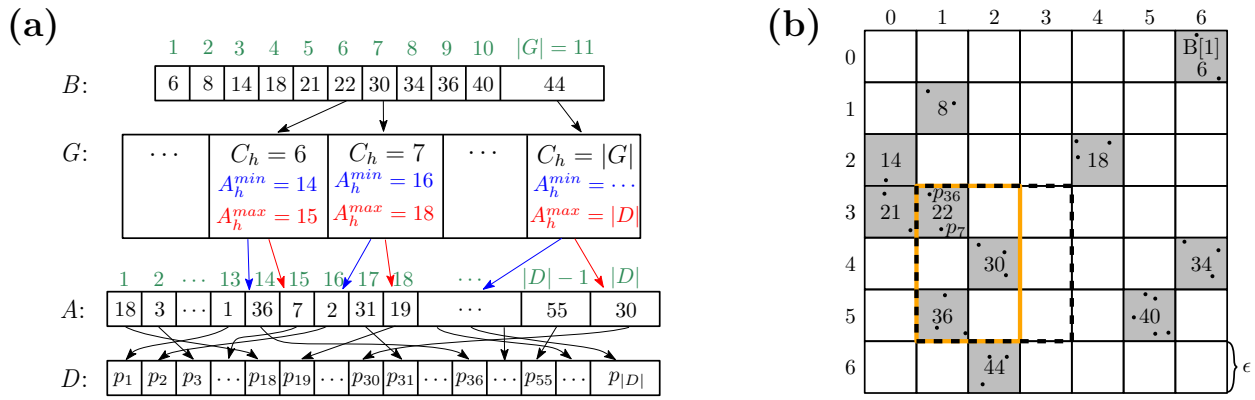


Figure 2: Example of our point indexing strategy. (a) contains the indexing structure we employ; and, (b) illustrates the resulting grid of points in 2-D space.

3.3. Index Components

In this section, we present the components of our grid index structure, illustrated with an example in Figure 2 (a). In the following section, we demonstrate how we search the structure using the example grid shown in Figure 2 (b). To minimize the space occupied by the index structure, we use several supporting components. Given a set of input data points, D, we create an index using a series of array structures, outlined in Table 1.

Based on the size and distribution of D and choice of ϵ, our grid index will have some number of non-empty grid cells. As illustrated in Figure 2 (a), each non-empty cell is stored in B with its linearized coordinate computed from the cell's n-dimensional coordinate in the grid. Each of these non-empty cells has a corresponding entry in G (i.e., $|B| = |G|$), which maps the cell to a contiguous set of points in A by storing the minimum and maximum index, A_h^{min} and A_h^{max}, respectively. Each entry in A corresponds to one n-dimensional data point from the original input set, D (i.e., $|A| = |D|$), allowing us to access the set of data points that are contained within a specific cell. The masking array, M, 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 $j \in \{1, \ldots, n\}$.

3.4. Index Search

Figure 2 (b) shows a 2-D example grid, where non-empty cells are shaded and are labeled in lexicographic order. In the example in the figure, there are 11 non-empty cells, so $|B| = |G| = 11$. We consider cell $C_h = 7$ (linear id 30) with point $a \in D$ and determine all neighboring points within ϵ of the adjacent grid cells. For each dimension, we begin by determining the range of neighboring cell coordinates, which we denote as O_j, where $j = 1, \ldots, n$. For cell $C_h = 7$, we have that $O_1 = [1, 3]$ and $O_2[3, 5]$ (cells outlined with the black dashed lines in Figure 2 (b)). We then compare O_j to the masking array M_j to eliminate empty rows and columns. In the example, this eliminates cells in the 3rd column, since there are no non-empty cells in that column. Thus, we have $O_1 \cap M_1 = [1, 2]$ and $O_2 \cap M_2 = [3,5]$ (6 cells outlined in orange in Figure 2 (b)). For each cell within the the range of each dimension (6 cells in the example), 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 out 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 $\{p_{36}, p_7\}$.

Table 1: An outline of the structures that our indexing scheme uses. For a given set of points, D, we create each of these structures.

Param.	Name	Description
B	Grid cell lookup	List of non-empty cells with linearized coordinates
G	Grid-point mapping	Mapping of non-empty cells to a range of points in A
A	Point lookup	List of points, ordered by grid cell with pointers to D
M	Masking array	For each dimension, list of coordinates with non-empty cells

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. Since GPUs have limited global memory (typically 4–16 GiB), this may result in an index that exceeds the maximum memory capacity. Another option for the data layout of the lookup array (A), would be to pad A such that the points in each cell are block aligned. However, ensuring block alignment will require additional space and memory management to pad non-full blocks. Because we aim to minimize the space complexity of the index, we elect to leave the points assigned to a cell in A unaligned. Therefore, the points in A are stored in contiguous space. With our strategy, the space complexity of the index structure is $O(|B| + |G| + |A|)$, where $|B| = |G|$ and $|A| = |D|$. Since our cells contain at least one point, this simplifies to $O(|D|)$.

The curse of dimensionality [11] is known to reduce the pruning efficiency of index searches in higher dimensions [14]. For each query cell, there are $3^n - 1$ adjacent cells, leading to

a significant number of cells that must be checked to determine if they exist in B (i.e., are non-empty). However, since average density decreases in higher dimensions, fewer adjacent cells to a query cell will be non-empty, allowing us to ignore more adjacent cells.

3.5. Global Memory GPU Kernel

In this section we provide details of GPUSelfJoinGlobal, 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. The GPU kernel takes as input D, A, G, B, M, and, ϵ (see Table 1 for parameter descriptions). Each thread begins by getting its global thread id (line 2), which is computed as follows: gid= threadIdx.x + (blockIdx.x*blockDim.x), where a single thread is assigned a single point, where the thread finds all of the neighbors of its assigned point. Thus, the global thread ids are in the range [0, $|D|$). Each thread checks to see if the global id is larger than $|D|$ (line 3), and if so, returns. 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 compute the linearized ids of the cells to query to be: 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
```

4. Optimizations

The indexing scheme presented in Section 3 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.

4.1. Batching the Result Set

Batching schemes are required by many GPU-accelerated approaches that process large volumes of data (e.g., when processing large matrices that exceed global memory capacity [33, 34, 35]). 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 [31, 36] that calculates the ϵ-neighborhood of 2-D points on the GPU for use in the DBSCAN clustering algorithm [1].

The total result set size of the self-join is data dependent, so we do not know how many batches must be executed for a given dataset and value of ϵ. If we denote $|R|$ as the total result set size, and each batch can store b_s result set elements, then the total number of batches is simply $n_b = \lceil |R|/b_s \rceil$. However, since we do not know the value of $|R|$, we cannot compute n_b (for now, we consider a fixed batch size, b_s, and discuss how it can be selected based on performance related factors in Section 5.4).

There are several options for accommodating non-deterministic and potentially large result set sizes on the GPU. One option is to allocate a result set buffer, and when an item is added to the buffer, we check for a buffer overflow. If buffer overflow would occur, the buffer contents are transferred back to the host, the buffer is cleared, and the kernel is re-attempted with the remaining queries [37, 7, 26]. Another option is to first execute the entire algorithm to count the total number of results that will be added to the result set (e.g., compute $|R|$), and then, from the exact value of $|R|$, compute the number of batches [38]. Each of these options have performance drawbacks: the latter option above simply computes double the total amount of work and the former option may also waste work. For instance, in the context of the self-join, if only a partial result can be stored in the result set for a given query point (i.e., all of a query point's neighbors cannot be stored in the result buffer before it would overflow), then this query point must be re-attempted in a subsequent

kernel invocation. This may also complicate processing the result set, as the result set may contain duplicates when query points fail to store all of their neighbors. We avoid both of these options for batching, and rely on a sampling approach that estimates n_b for a given b_s.

We summarize our batching scheme and refer the reader to [31, 36] for more information. We incrementally process the dataset by executing multiple kernels, where each kernel processes a fraction of the entire dataset, D. First, we execute a kernel that calculates the number of neighbors within ϵ for a fraction, $f = 0.01$, of the points, denoted as a_b. Then, we estimate the total size of the result set as $e = (1/f)a_b \times 1.05$. This overestimates the result set size by approximately 5% to decrease the probability that the result set buffer for each batch overflows (using this value, the buffer never overflows in our experimental evaluation). Given a buffer size on the host for a single batch, b_s, the total number of batches is $n_b = \lceil e/b_s \rceil$. To balance the workload of each batch, we assign data points in similar spatial regions to *separate* batches, balancing the result set size of each batch. This avoids having one batch being assigned all points in a dense spatial region, which would result in a large result set that could overflow the buffer. After the results are sent to the host, the data is copied from a *pinned memory* staging area (for faster data transfers from the device, and to allow overlapping of data transfers in streams), and pointers are set to map a given point to its neighbors that are within the ϵ distance. Pinned memory is memory allocated on the host that allows for data to be directly transferred to and from the memory location by the GPU (see CUDA documentation for details [39]).

As mentioned above, the number of points that are sampled to compute the estimate of the result set size, e, is selected to be $f = 0.01$. Increasing f yields a more accurate estimate of e, but increases the associated overhead. Note, however, that the number of batches is computed using the ceiling, as we cannot have a non-integral number of batches. Therefore, a very accurate estimate is not required, as the number of batches is rounded up, which decreases the chances of buffer overflow. Furthermore, the computation of e includes an overestimation of 5% to ensure that buffer overflows do not occur. This overestimation factor should not be significantly increased, as it would increase the total number of batches, resulting in performance loss due to a large number of small batches being executed on the GPU. However, as shown in our performance model (Section 5.4), small increases in the number of batches are not expected to significantly degrade performance.

There are several performance advantages to the batching scheme. All batches are independent of each other, so we can pipeline several operations to maximize resource utilization on the host and GPU. In particular, while the dataset and index reside on the GPU across all kernel invocations, we can execute the following operations concurrently: (*i*) sending a batch's kernel parameter data to the GPU (host-to-device); (*ii*) computation of a kernel on the GPU; (*iii*) result set data transfer to the host (device-to-host); and (*iv*) performing host-side operations, such as memory copies between pinned memory buffers and the pageable memory buffers (since we use the pinned memory buffers as a staging area). This pipelining partially mitigates overheads such as data transfers, which are a common bottleneck in GPU computing [40]. Thus, even for workloads with result sets that would not overflow the GPU's global memory capacity, it is more efficient to partition the work into smaller chunks and use the batching scheme to concurrently execute the four major operations above.

In all experiments where we make a comparison to other approaches, we use 3 CUDA streams (i.e., $n_b \geq 3$) and the batch size, b_s, is set to 5×10^7. In the experimental evaluation in Section 5 we demonstrate the relationship between batch size and performance, and we present a performance model that enables us to select a value of b_s that improves overall performance.

4.2. Avoiding Duplicate Calculations

Euclidean distance is reflexive, so, for two points $p, q \in D$, if p is within ϵ-distance of q, then q is within ϵ of p. Thus, if we can evaluate pairs of neighboring points only *once*, we can report both corresponding ordered pairs and eliminate redundant work. Using our linearization of grid cells described in Section 3, we can avoid duplicate comparisons by only comparing each point to points in neighbor cells with a higher linear ID. However, this method, which we call "linearized coordinates" (LINEARCOORD), requires additional work (computing the linear ID) to be performed for every non-empty adjacent cell. As dimensionality increases, this may result in a significant overhead. Thus, we present another general method of accomplishing this for n dimensions, which we call "uni-directional comparison" (UNICOMP).

The UNICOMP optimization operates as follows: 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.

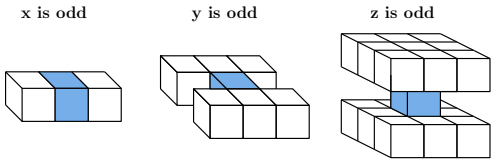


Figure 3: Illustration of the UNICOMP optimization. The blue cell is our source cell and the white cells drawn around it are those we want to evaluate if the source cell index at the particular dimension is odd.

In the example of $n = 3$ (x, y, and z coordinates), with UNICOMP 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 same z-index as C_a, and, if even, we do nothing. 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 note that if a cell has all even coordinates,

it is completely ignored and does not require any processing. 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.

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.z 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
```

Both UNICOMP and LINEARCOORD reduce the number of distance calculations by a factor of two. While UNICOMP also reduces the number of cell evaluations by a factor of two, it is more complex and requires additional computation to determine which cells can be avoided. LINEARCOORD, however, is very simple, though it must compute the linear ID of every adjacent cell to determine if it must be evaluated or not. Thus, it is not clear which approach is better in practice, so we experimentally compare the performance impacts of each in Section 5.2.1.

5. Experimental Evaluation

5.1. 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 we discuss in Section 5.3.3, uniformly distributed data results in worst-case performance for our GPU grid indexing scheme because it 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 synthetic datasets using a uniform distribution, generated in the range [0,100] in each dimension in 2–6 dimensions with 2 and 10 million points to observe how performance varies as a function of data density and dimensionality. To assess how performance changes with data distribution, we also generate two exponentially distributed datasets (2-D and 6-D) with $\lambda = 40$. Additionally, we evaluate performance using three types of real-world datasets. The *SW-* dataset contains the latitude/longitude and the total electron content in the ionosphere [41]. 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 [42]. Galaxies are represented in 2-D (right ascension and declination), spanning redshift (z) of $0.30 \leq z \leq 0.35$. Finally, we evaluate performance using the *ColorHist* dataset, which contains 32-D image features, and 68,040 points, obtained from the UCI machine learning repository [43]. While the dimensionality of *ColorHist* is beyond the dimensionality that we consider in this work, we use it as a diagnostic to evaluate the scalability of SUPEREGO and ensure consistency between the SUPEREGO implementation and the paper that presents the algorithm [8]. A summary of dataset properties are outlined in Table 2.

Table 2: 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
SynExpo2D2M	2×10^6	2	*SynExpo6D2M*	2×10^6	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
Real World Diagnostic:			*ColorHist*	68,040	32

5.2. Experimental Methodology

We experimentally evaluate the performance of four different implementations: our grid-based GPU implementation (GPU-SJ), a sequential reference implementation (CPU-RTREE), the state-of-the-art SUPEREGO implementation (SUPEREGO), and a brute-force GPU implementation. All GPU code is written in CUDA [39], and all C/C++ host code is compiled with the GNU compiler with the O3 optimization flag. Details about each implementation are presented in this section.

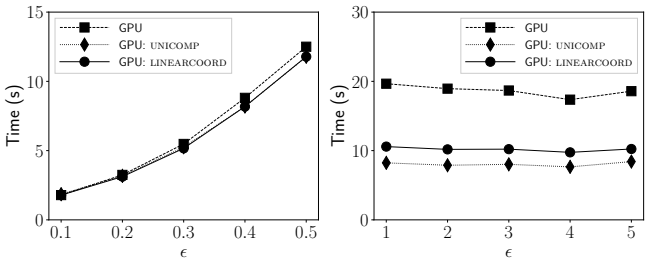
We use two hardware platforms for experimental evaluation, detailed in Table 3. We note that PLATFORM1 has a more powerful CPU (and twice and many CPU cores). Therefore, for the majority of our experiments, we execute GPU-SJ and SUPEREGO on PLATFORM1 to give an advantage to the state-of-the-art CPU algorithm, SUPEREGO. If we were to compare GPU-SJ and SUPEREGO on PLATFORM2, then we would be giving an advantage to GPU-SJ. Thus, we only use PLATFORM2 for a subset of the experiments (such as determining if there is significant performance degradation by using 64-bit vs. 32-bit floats). All experimental results are averaged over 3 trials. We exclude the time to construct the GPU-SJ grid index because we also exclude index construction time for the other implementations with which we compare performance (CPU-RTREE and SUPEREGO). Since these other implementations do not optimize index construction, it is not fair to compare performance by including index construction time. However, we note that one

Table 3: Details of hardware platforms and floating point precision of execution.

	CPU				GPU				
Platform	Model	Cores	Clock	Memory	Model	Cores	Memory	Software	Precision
Platform1	2×Xeon E5-2683 v4	$2 \times 16 = 32$	2.1 GHz	256 GiB	Titan X	3584	12 GiB	CUDA 9	64-bit
Platform2	2×Xeon E5-2620 v4	$2 \times 8 = 16$	2.1 GHz	128 GiB	Quadro GP100	3584	16 GiB	CUDA 9	32-bit, 64-bit

advantage of CPU-RTree is that it can be queried for any value of ϵ, whereas GPU-SJ and SuperEGO require reconstructing the index for each value of ϵ.

The selection of ϵ determines the size of the neighborhood to search, and thus the average number of neighbors found within ϵ of each point. Selecting too small of a value of ϵ will find no neighbors, and selecting too large of a value of ϵ may find too many neighbors (i.e., each point could find all of the points in the dataset, or $|D|$ neighbors). Such ranges (too many or too few points) are not a pragmatic use case for the self-join. Thus, in our results, we present the ranges of the average number of neighbors per point. This demonstrates that we find a moderate number of neighbors, and this information can be used in reproducibility studies and comparisons to our work.



(a) *Syn2D10M* (b) *Syn6D10M*

Figure 4: Average response time with and without each optimization to avoid duplicate calculations on Platform2. The UNICOMP optimization provides the largest performance gains due to reduced overhead.

5.2.1. Our Grid-based GPU Implementation (GPU-SJ)

Our GPU self-join implementation, GPU-SJ, implements the GPU grid index structure described in Section 3 and the optimizations presented in Section 4. For all experiments, we execute the GPU-SJ self-join CUDA kernel (Algorithm 1) with 256 threads per thread-block. As discussed in Section 4.1, we use 3 CUDA streams and a batch size of $b_s = 5 \times 10^7$ for all experiments, unless otherwise mentioned. Our GPU-SJ implementation supports both 32-bit and 64-bit floats (although we test SuperEGO with 32-bit floats), and we measure the associated performance impacts in Section 5.3.7.

To determine whether to use the LINEARCOORD or UNICOMP optimization to avoid duplicate distance calculations (described in Section 4.2), we experimentally measure the performance impact of each. Figure 4 plots the average response time of GPU-SJ with the LINEARCOORD optimization, the UNICOMP optimization, and neither (no duplicate elimination), on synthetic datasets of 10 million points. The results in Figure 4 indicate that, while both UNICOMP and LINEARCOORD reduce overall response time, they only provide marginal performance gains on low dimensional datasets (Figure 4 (a)). In higher dimensions, however, they both improve performance significantly, with UNICOMP providing the largest performance gains. We attribute this to the larger overhead associated with the LINEARCOORD optimization computing the linear ID of every adjacent cell. Thus, in all experiments that follow, we either use no duplicate avoidance optimization or we use the UNICOMP optimization. We note that there may be some instances where LINEARCOORD can outperform UNICOMP, for instance, in higher dimensions than that considered in this work, but we do not observe this behavior.

5.2.2. Sequential 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 [44]). 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 less work than constructing the R-tree.

The index search-and-refine strategy is particularly efficient at low-dimensionality [14]; however, at higher dimensions, index searches become less effective at pruning the search for nearby neighbors. This phenomena is referred to as the curse of dimensionality [11, 12, 13]. For this reason, *we only focus on dimensions 2–6*, and do not claim that our methods are directly applicable at higher dimensionality.

5.2.3. Parallel State-of-the-art (SuperEGO)

The SuperEGO algorithm [8] (overview provided in Section 2.2) performs fast self-joins on multidimensional data. The parallel CPU implementation has been shown to outperform many other join algorithms on both low and high dimensional data and is considered state-of-the-art. When comparing our approach, GPU-SJ, to the multi-threaded version of Super-EGO [8], we use 32 threads on our 32 core platform (Platform1). We execute the algorithm using 32-bit floats (execution with 64-bit floats failed). 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 validate consistency between our implementations by comparing the total number of neighbors within ϵ. In all runtime measurements used to compare approaches, we use the total time to *ego-sort* (the operations used to create the index structures) and perform the join. We are grateful to D. Kalashnikov for making his code

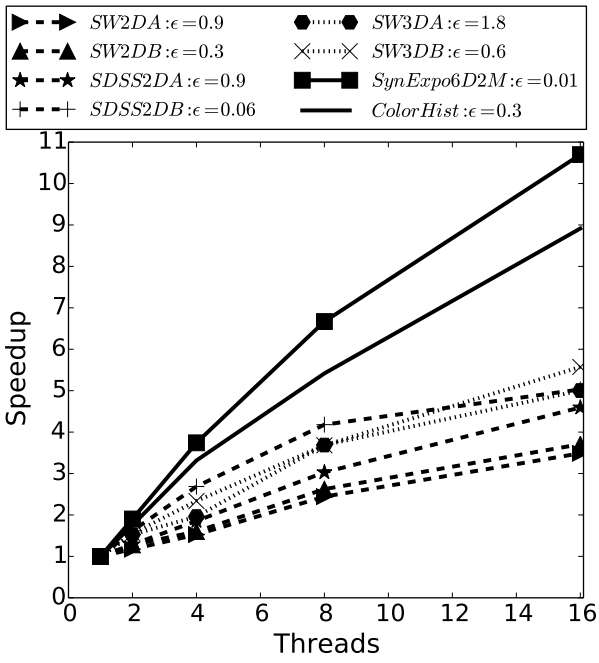


Figure 5: Scalability of SuperEGO on the real-world datasets in 2-D and 3-D. SuperEGO has poor scalability on low-dimensional datasets. As a diagnostic to ensure that we can reproduce the scalability results in [8], we execute SuperEGO on *ColorHist* and find that the scalability results are consistent (see Figure 29 in [8]). Execution performed on Platform2.

publicly available.

Before comparing our approach to SuperEGO, we benchmark the scalability of SuperEGO on Platform2. We first benchmark SuperEGO to determine if it can achieve the greatest performance when using all available cores. In this benchmark, we only include the time to execute the parallel section of the algorithm (the time to join), and exclude the time to egosort. From Figure 5, we find that even the parallel portion of SuperEGO has poor scalability on some of the low-dimensional datasets. SuperEGO achieves the best scalability on the 6-D dataset, *SynExpo6D2M*. Furthermore, performance scales better with the 3-D datasets than the 2-D datasets, indicating that the scalability of SuperEGO may be dependent on dimensionality. To ensure that our results are consistent with those presented in the work of Kalashnikov [8], we execute SuperEGO on the dataset that they use to assess scalability, *ColorHist* (see Figure 29 in that paper). We find that when we execute SuperEGO, the scalability of the 32-D dataset, *ColorHist*, is consistent with their results. Thus, we conclude that, at least for the datasets used in our work, the scalability of SuperEGO is dependent on dimensionality. Furthermore, SuperEGO can efficiently use all hardware cores, so we use P threads when executing SuperEGO, where P is the number of CPU processor cores on the hardware platform.

5.2.4. GPU Brute Force Implementation

As dimensionality increases, the overhead of searching an index structure increases. Thus, for datasets of large enough 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 compare the performance of our approach, GPU-SJ, to a parallel brute force implementation on the GPU. The kernel simply uses $|D|$ threads, and each thread is assigned one point to compare to every other point 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 response time of the brute force implementation, as it only considers the time spent computing the self-join. 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. Since this brute force implementation executes on the GPU, it demonstrates that the performance gains of our approaches are *not* solely due to the high throughput of the GPU.

5.3. Results

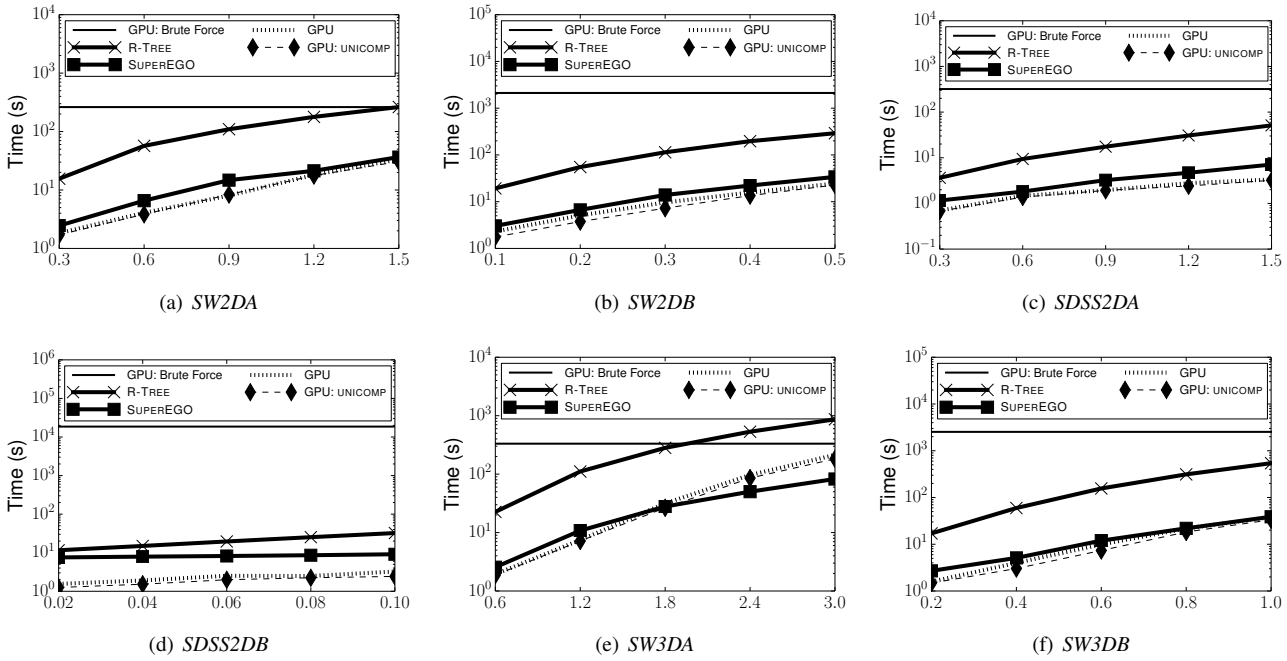
As the search distance, ϵ, increases, the performance of the self-join degrades for two main reasons: (*i*) more candidates will be found, thus requiring more distance calculations to determine if points are within ϵ distance of each other; and, (*ii*) the performance of index searches degrade as the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

5.3.1. Real-world Datasets

Figure 6 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. However, on the largest dataset, *SDSS2DB* (containing 15 million points), GPU-SJ (with unicomp) achieves up to ~6× speedup over SuperEGO. Although GPU-SJ is faster than the brute force method for all of our experiments, on the *SW3DA* dataset (Figure 6 (e)), brute force outperforms CPU-RTree at $\epsilon > 1.8$. This is because the ϵ values are quite large in this experiment, resulting in a large number of distance calculations and a relatively ineffective index.

5.3.2. Synthetic Datasets

Figures 7 and 8 show the response time vs. ϵ for synthetic datasets with 2 and 10 million data points, 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 6, 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 7 and 8 (b)-(e)), but it does not yield a factor of 2 reduction in response time across all scenarios as expected. Across all



(a) *SW2DA* (b) *SW2DB* (c) *SDSS2DA*

(d) *SDSS2DB* (e) *SW3DA* (f) *SW3DB*

Figure 6: 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. The rounded average number of neighbors per point are as follows: (a) 296–5.8k, (b) 92–2.0k, (c) 36–851, (d) 2–33, (e) 240–13.2k, (f) 34–2.1k. Execution performed on PLATFORM1.

synthetic datasets, GPU-SJ (with UNICOMP) outperforms CPU-RTREE. Furthermore, GPU-SJ with UNICOMP outperforms SUPEREGO on nearly all experiments.

We note that all algorithms (except for brute force) are significantly faster when $n = 3$, as compared to $n = 2$. This is due to the decrease in point density with increased dimension, as observed in Figure 1 (a). The exception is at $\epsilon < 3$ on *Syn6D10M* (Figure 8 (e)), where the performance is roughly independent of ϵ. For all experiments, the GPU brute force implementation is at least an order of magnitude slower than GPU-SJ.

5.3.3. 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).

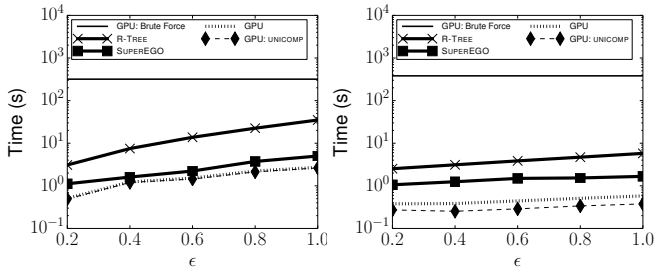
SUPEREGO also tends to perform better on non-uniform data for many of the same reasons as GPU-SJ. However, SUPEREGO reorders the dimensions to improve cell pruning as a function of data distribution [8]. We find that across the uniformly distributed synthetic datasets, the performance of SUPEREGO degrades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

To determine how GPU-SJ performs on skewed data distributions, we plot the response time vs. ϵ for two synthetic datasets with an exponential distribution in 2-D and 6-D in Figure 9. We do not plot the R-tree and brute force methods so that we can show the plot on a linear scale. We find that, on exponential datasets, the performance of GPU-SJ with UNICOMP yields similar performance gains over SUPEREGO as on the uniformly distributed datasets. On the *SynExpo2D2M* dataset the speedup of GPU-SJ over SUPEREGO is between 1.72-2.94× and on *SynExpo6D2M* it is between 2.35-2.86×. Note that this experiment was executed on PLATFORM2, where we have 16 and not 32 cores as on PLATFORM1. However, the scalability of SUPEREGO on low dimensional datasets is low (Figure 5); therefore, we do not expect significant performance loss due to the execution of SUPEREGO on PLATFORM2.

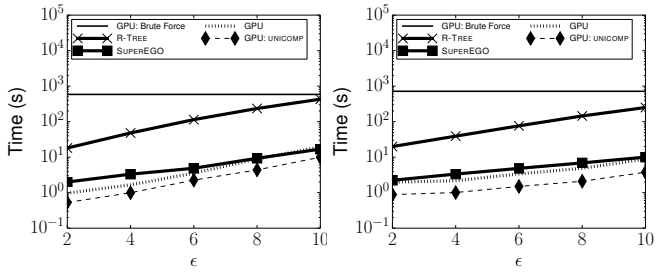
5.3.4. Batch Size and GPU Occupancy

Modern GPUs have many compute cores, so they require a large number of threads to achieve peak performance. Additionally, GPUs use fast context switching and oversubscription to hide memory latency and maximize computational throughput by assigning multiple threads to each compute core [45]. *Occupancy* measures the number of threads assigned to each core, as a percentage maximum possible for the GPU hardware. Factors such as shared memory usage, branch divergence, and available parallelism impact the occupancy. Thus, occupancy is a useful metric to determine application efficiency on the GPU. For more details on occupancy and its impact on GPU performance, see standard documentation [39].

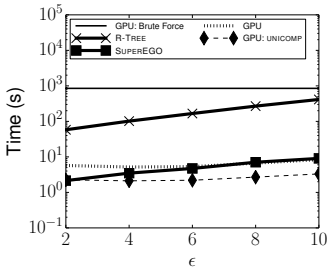
Recall that the batching scheme described in Section 4.1 al-



(a) *Syn2D2M* (b) *Syn3D2M*

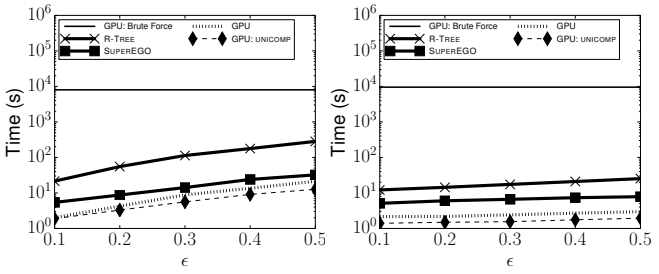


(c) *Syn4D2M* (d) *Syn5D2M*

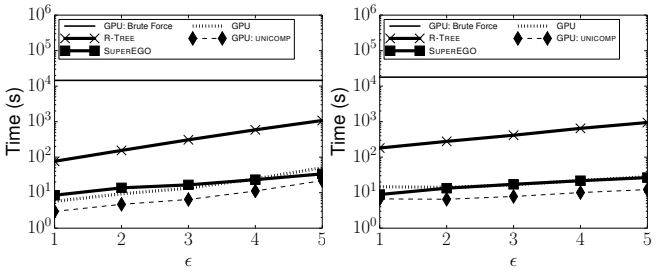


(e) *Syn6D2M*

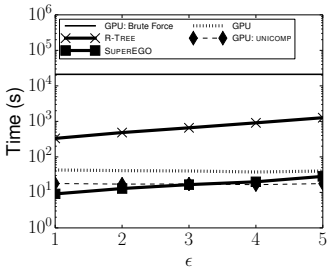
Figure 7: Response time vs. ϵ on the 2–6 dimensional synthetic datasets with 2×10^6 points. The rounded average number of neighbors per point are as follows: (a) 26–624, (b) 1–9, (c) 3–860, (d) 1–91, (e) 1–10. Execution performed on Platform1.



(a) *Syn2D10M* (b) *Syn3D10M*



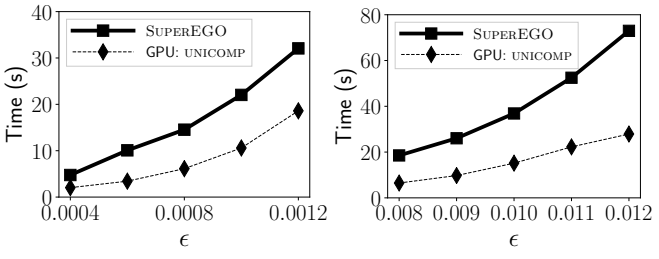
(c) *Syn4D10M* (d) *Syn5D10M*



(e) *Syn6D10M*

Figure 8: Response time vs. ϵ on the 2–6 dimensional synthetic datasets with 10^7 points. The rounded average number of neighbors per point are as follows: (a) 32–783, (b) 1–6, (c) 1–289, (d) 1–16, (e) 1–2. Execution performed on Platform1.

lows us to process large result sets, as well as perform concurrent operations, such as data transfers and computation. However, using many small batches results in more overhead, less parallelism per kernel, and potentially higher branch divergence. Thus, to determine how the number of batches, n_b, and occupancy impact overall performance, we measure occupancy and response time for varying values of n_b. Table 4 shows the batch size (b_s), number of streams, the total number of batches (n_b), the achieved occupancy, and the response time on the *SynExpo2D2M* dataset with $\epsilon = 0.0004$ when using UNICOMP. The theoretical maximum achievable occupancy on this dataset is 75%. As illustrated in Table 4, the achieved occupancy is greatest when $n_b = 1$, and that the worst achieved occupancy is when we have the largest number of batches, $n_b = 17$. By using more batches, we execute fewer threads per kernel invocation, and the data points assigned to each block are more dispersed, causing the threads in a warp to be less likely to have similar execution pathways. Therefore, an increase in n_b decreases the efficiency of the kernel. However, as seen in Table 4, the lowest



(a) *SynExpo2D2M*, $|D| = 2 \times 10^6$. (b) *SynExpo6D2M*, $|D| = 2 \times 10^6$.

Figure 9: Response time vs. ϵ for SuperEGO and GPU-SJ with UNICOMP on the synthetic exponentially distributed datasets in (a) 2-D and (b) 6-D. Execution performed on Platform2. SuperEGO is executed with 16 threads on 32-bit floats. GPU-SJ is executed with 64-bit floats. The rounded average number of neighbors per point are as follows: (a) 397–3.5k, (b) 100–863.

response time is achieved with the smallest batch size (with the lowest occupancy). This is because increasing the number of batches also increases the number of concurrent operations, as

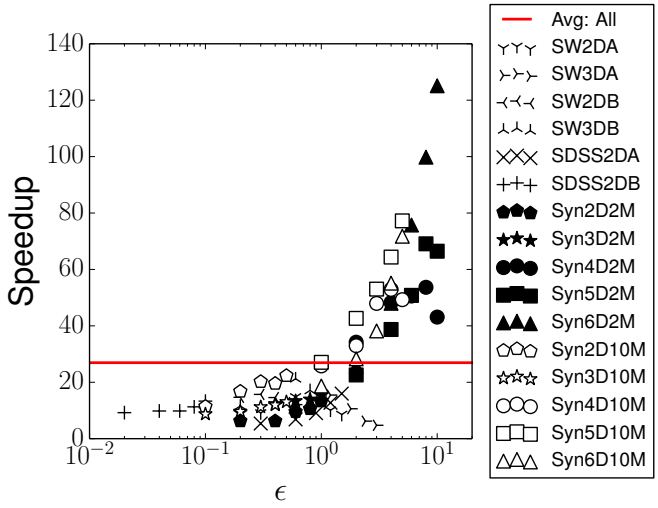


Figure 10: Speedup of GPU-SJ with UNICOMP over CPU-RTREE (1 thread) for real-world and synthetic datasets (derived from Figures 6, 7, and 8) plotted on log scale to capture orders of magnitude differences in ϵ. The horizontal line shows the average speedup (26.9×).

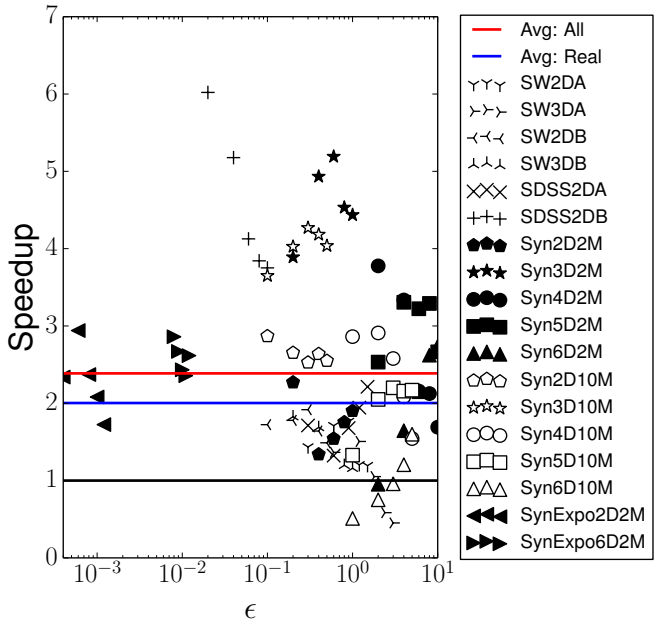


Figure 11: Speedup of GPU-SJ with UNICOMP over SUPEREGO for all real-world and synthetic datasets plotted on log scale to capture orders of magnitude differences in ϵ. Figure derived from Figures 6, 7, 8 and 9. Horizontal lines from top to bottom: the average speedup across all datasets (top, red), average speedup across all real-world datasets (middle, blue), demarcation of speedup or slowdown (bottom, black).

discussed in Section 4.1. Furthermore, larger batch sizes incur more pinned memory allocation overhead, which is why, when we execute a single batch in a single stream with $b_s = 9 \times 10^8$, the response time is the greatest. In Section 5.4 we develop a performance model that allows us to select a good batch size, partially as a function of the pinned memory allocation overhead. The results in Table 4 demonstrate that occupancy does not have as large of an impact on the performance of GPU-SJ as performing concurrent operations. Therefore, we trade occupancy for increased concurrency (e.g., overlapping data transfers, and computation on the CPU and GPU) to achieve good performance.

5.3.5. *Summary of Performance*

GPU-SJ vs. CPU-RTREE: Figure 10 plots the speedup of GPU-SJ with UNICOMP over CPU-RTREE vs. ϵ for each dataset from Figures 6, 7, and 8. 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 10) 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 dimensional datasets (up to 125×). Across all datasets, GPU-SJ is on average 26.9× faster than CPU-RTREE.

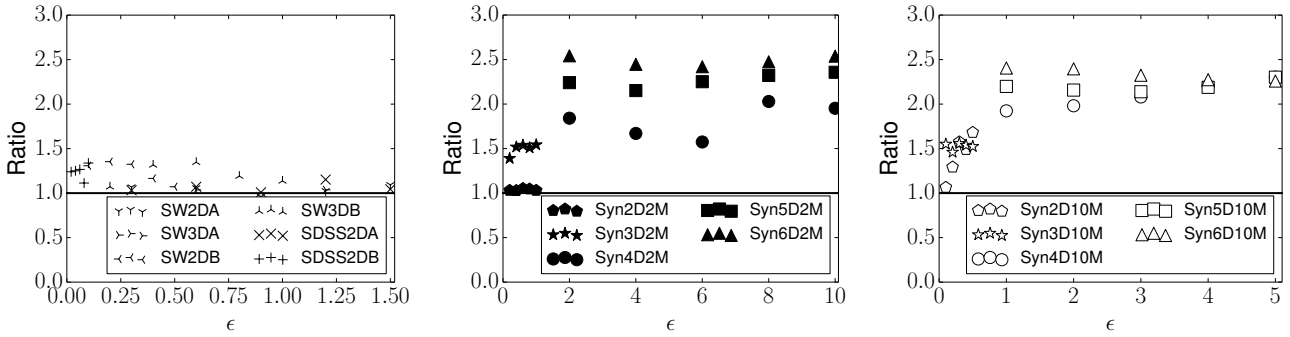
GPU-SJ vs. SUPEREGO: Figure 11 plots the speedup of GPU-SJ with UNICOMP over SUPEREGO (derived from the respective figures). Similar to Figure 10, the GPU performance gain is lowest on scenarios with small workloads. We find that there are only 6 instances where SUPEREGO outperforms GPU-SJ (below the bottom horizontal line). The average speedup on real-world datasets is ∼ 2× (middle horizontal line), while the average across all datasets is 2.39× (top horizontal line). As expected, SUPEREGO performs worse on the synthetic datasets, as it cannot benefit from dimensionality reordering on uniformly distributed data. While SUPEREGO is a state-of-the-art parallel CPU join algorithm, across nearly all scenarios, our GPU-SJ algorithm outperforms SUPEREGO.

Table 4: The relationship between occupancy and the number of batches. GPU-SJ with UNICOMP is executed on *SynExpo2D2M* and uses 64-bit floats and $\epsilon = 0.0004$. The theoretical occupancy of the kernel is 75%. The achieved occupancy shown is measured as the first batch in the execution of GPU-SJ (all batches have nearly equal work, and have the same performance characteristics). Execution performed on PLATFORM2.

b_s	Streams	# Batches (n_b)	Achieved Occupancy (%)	Times (s)
5×10^7	3	17	57.8	2.07
1×10^8	3	9	60.9	2.71
2×10^8	3	5	63.6	4.01
3×10^8	3	3	64.9	5.07
9×10^8	1	1	65.9	6.30

5.3.6. *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



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

Figure 12: 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 6, 7, and 8, respectively. Points above the horizontal line indicate that UNICOMP leads to a performance gain, whereas points below indicate a slowdown due to associated overheads. Execution performed on PLATFORM1.

Table 5: 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. Execution performed on PLATFORM1.

Dataset	ϵ	Ratio Resp. Time	Theoretical Occupancy			Cache Bandwidth Utilization (GB/s)		
			No UNICOMP	UNICOMP	Ratio	No UNICOMP	UNICOMP	Ratio
SW2DA	0.3	1.07	100%	75%	0.75	597.87	460.96	0.77
SDSS2DA	0.3	1.03	100%	75%	0.75	1157.07	864.26	0.75
Syn5D2M	8	**2.32**	62.5%	50%	0.8	306.45	578.14	**1.88**
Syn6D2M	8	**2.47**	62.5%	50%	0.8	293.31	469.24	**1.59**

as a function of ϵ and dataset size, as mentioned above. Figure 12 shows the ratio of the response times of GPU-SJ with and without UNICOMP, derived from the results shown in Figures 6, 7, and 8 (by dividing the respective response times). While there are are a few scenarios in Figure 12 (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 12 (a)), the response time ratios are within 1.5×, and not 2× as expected. However, the higher dimensionality ($n \ge 3$) results shown in Figure 12 (b) and (c) demonstrate that UNICOMP achieves some performance gains that are ≥ 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 [46] to collect two metrics during execution: *occupancy* and *unified cache utilization*. As discussed in Section 5.3.4, occupancy is a measure of the number of threads running on the hardware, and low occupancy can result in under-utilization of the GPU. On Maxwell and Pascal generation NVIDIA GPUs, the unified (L1) cache is a coalescing buffer for memory accesses [47]. 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 5 details the occupancy and unified cache bandwidth utilization when running GPU-SJ (with and without UNICOMP) for data sets with response time ratios < 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. 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.

5.3.7. Performance of 32-bit vs. 64-bit Floats

As mentioned in Section 5.2, we executed SUPEREGO with 32-bit floats and GPU-SJ with 64-bit floats. This favors SUPEREGO, as 64-bit floating point operations are more expensive than 32-bit floats, and GPUs have more hardware dedicated to 32-bit operations [15]. On PLATFORM2, we execute all of the experiments shown in Figures 6, 7, 8 and, 9 for both 32-bit and 64-bit floats. The ratio of the response time of 64-bit floats to 32-bit floats is shown in Figure 13. At most, the 32-bit floating point execution is a factor ∼ 1.2 faster than 64-bit floats. However, overall, there is very little performance loss from a 64-bit execution. We attribute this to the fact that the search phase of the algorithm performs operations primarily on integers. Therefore, there is not a sufficiently high ratio of floating point operations to integer operations, such that we would observe a large performance gain from computing the self-join using lower precision (32-bit) coordinates. However, the self-join on higher dimensional data would increase the number of

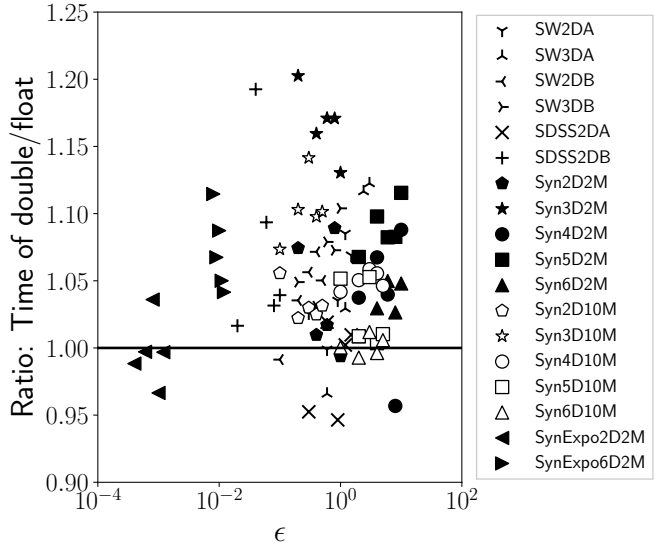
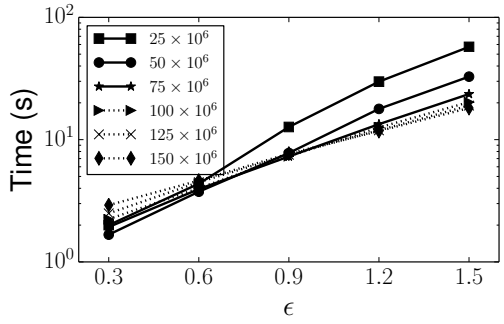


Figure 13: Ratio of the response time to execute GPU-SJ with UNICOMP using doubles to floats. All of the experimental scenarios in Figures 6, 7, 8 and 9 are shown, except that the execution is performed on PLATFORM2. The black line shows where floats and doubles achieve the same performance.

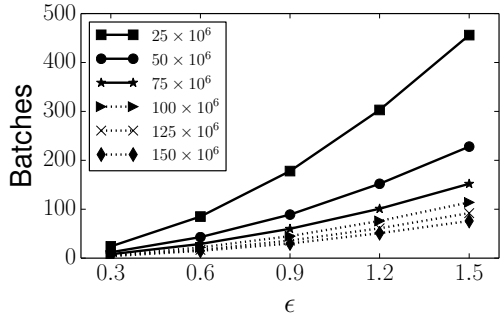
operations needed to compute the distance between two points; therefore, we may expect 32-bit operations to lead to larger performance gains on higher dimensional data. Also, note that some of the ratios are below the horizontal line that denotes a slowdown. This is due to variance in measurements on small workloads, where we notice that the scheduler will sometimes overlap kernel executions. On small workloads, this adds variance to the time trials (some of the executions require a time $\ll 1$ s to execute). In summary, we do not find that GPU-SJ would further gain a significant performance advantage (e.g., by a factor > 2) over SUPEREGO if we executed GPU-SJ using 32-bit floats. Thus, the results in Figure 11 (and related figures), correspond to an approximate lower bound on the performance gain over SUPEREGO.

5.4. Performance Model: Optimizing the Batch Size

As mentioned in Section 4.1, we execute GPU-SJ with 3 CUDA streams, allowing us to overlap kernel invocations and data transfers between the host and GPU in addition to performing other concurrent host-side tasks. Thus, GPU-SJ executes in a series of n_b batches, each of size b_s. In all of the results shown in our evaluation, we use a batch size of $b_s = 5.0 \times 10^7$, although there is a performance trade-off between using too few and too many batches. If b_s is large, there is more pinned memory allocation overhead, while if b_s is small, there is more overhead associated with invoking so many kernels. Furthermore, a small batch size means that there are fewer total threads being executed per kernel invocation, which can result in GPU underutilization. b_s. Therefore, ideally, b_s should be carefully selected given the input dataset and ϵ to minimize overall GPU-SJ response time. We did not vary b_s in the evaluation thus far, demonstrating that our approach outperforms the state-of-the-art without excessive parameter tuning. In this section, we



(a) Response time vs. ϵ



(b) Number of batches, n_b, vs. ϵ

Figure 14: Comparison of performance as a function of batch size, b_s, on *SW2DA*. Execution performed on PLATFORM2.

advance a performance model that can be used to predict the response time of GPU-SJ as a function of b_s, thereby allowing us to select a value of b_s that results in good performance. By using a geometric formulation of the expected workload (the total result size), our performance model relies on only a few parameters and empirical measurements.

We execute GPU-SJ on *SW2DA* for the following values of b_s: (25, 50, 75, 100, 125, 150)$\times 10^6$. Figure 14 shows the relationship between performance (response time) and the number of batches executed. From Figure 14 (a) on the smallest workload ($\epsilon = 0.3$), we find that the smallest batch size, $b_s = 25 \times 10^6$, achieves the best performance, whereas the largest batch size, $b_s = 150 \times 10^6$ achieves the worst performance. This is because the pinned memory allocation overhead of the largest value of b_s dominates the response time. For the largest workload, $\epsilon = 1.5$, the largest batch size, $b_s = 150 \times 10^6$, achieves the best performance. From Figure 14 (b) we see that at $\epsilon = 1.5$ and $b_s = 25 \times 10^6$, 456 kernel invocations are required, leading to significant overhead, in comparison to fewer than 100 kernel invocations at $b_s = 150 \times 10^6$. For ϵ values between these extremes, we observe that intermediate values of b_s are needed to minimize response time.

Without the overhead of pinned memory allocation, using a larger batch size will outperform a smaller batch size because there are fewer kernel invocations and each invocation has a performance cost. Figure 15 plots the response time vs. the number of batches for $\epsilon = 0.9$, which shows that pinned

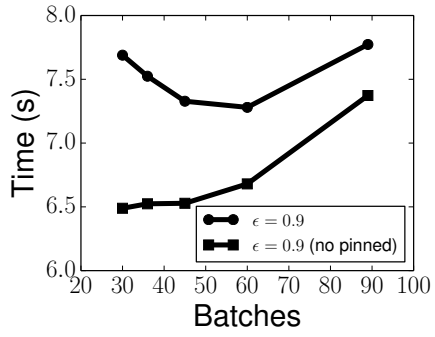


Figure 15: Response time vs. number of batches, n_b, on *SW2DA* with ϵ = 0.9 with the pinned memory allocation both included and removed from the response time. For illustrative purposes, the number of batches is limited to $n_b < 100$. Execution performed on Platform2.

memory allocation impacts performance when the number of batches is small (i.e., the batch size, b_s, is large). If pinned memory allocation was negligible, then we would simply execute GPU-SJ with the largest batch size that would fit into global memory (while still using 3 streams to exploit concurrency on the host and GPU). However, we observe that these overheads are non-negligible, and therefore, b_s must be carefully selected to mitigate these overheads.

5.4.1. Model Parameters

We model the response time of GPU-SJ on a given input as the sum of: (*i*) the time to execute all kernels/batches on the GPU, (*ii*) the time spent allocating pinned memory buffers, and (*iii*) the overheads associated with running multiple kernels. We define the following parameters that we use to model the total response time.

- $T_1(\epsilon)$ – the total time to execute GPU-SJ, for search radius ϵ, excluding the time needed for pinned memory allocation and kernel invocation overhead,
- $n_b(b_s, \epsilon)$ – the number of batches executed by GPU-SJ for batch size b_s, and search radius ϵ,
- $T_2(b_s)$ – the time needed to allocate a pinned memory buffer of size b_s on the host, and
- θ – the overhead of executing a single kernel.

Using these parameters, the total response time is:

$$R = T_1(\epsilon) + 3T_2(b_s) + \theta \cdot n_b(b_s, \epsilon), \quad (1)$$

where the factor 3 is due to our using 3 CUDA streams, each requiring a pinned memory buffer of size b_s.

These parameters are all platform/application dependent and we measure them as follows. We first compute $T_2(b_s)$ by executing a simple benchmark that measures the time to allocate pinned memory and, assuming that allocation time scales linearly with buffer size, we can then compute T_2 for any batch size. To measure the other parameters, we select one value of ϵ and execute GPU-SJ on the input data for two different batch sizes, b_s^α and b_s^β. We denote the GPU-SJ response time for b_s^α and b_s^β as R^α and R^β, respectively (corresponding to Equation 1). Hereafter, for simplicity we refer to $n_b(b_s^\alpha, \epsilon)$ and $n_b(b_s^\beta, \epsilon)$ as n_b^α and n_b^β, respectively. Using the results of this execution, we compute the overhead per kernel invocation/batch by taking the difference between the response times (with the pinned memory allocation time removed), divided by the difference in the number of batches, i.e.,

$$\theta = \frac{|(R^\alpha - 3T_2(b_s^\alpha)) - (R^\beta - 3T_2(b_s^\beta))|}{|n_b^\alpha - n_b^\beta|}. \quad (2)$$

We can then compute the response time, excluding pinned memory allocation and kernel invocation overhead as $T_1(\epsilon) = R^\alpha - 3T_2(b_s^\alpha) - \theta \cdot n_b^\alpha$.

5.4.2. Estimating Response Time

To analytically determine the best batch size for a given search radius, we use the parameters (defined above) to estimate the response time of GPU-SJ for any batch size b_s and search radius ϵ.

Assume that the search radius and batch size used to measure the above parameters are ϵ^α and b_s^α, respectively (i.e., we compute the relevant parameters, $T_1(\epsilon^\alpha)$, etc. in Section 5.4.1), and we wish to compute the response time for a search radius of ϵ and batch size of b_s. We denote the ratio of the volume (area in 2-D, as presented in our validation case) of the two ϵ search radii as: $V(\epsilon^\alpha, \epsilon) = \frac{\epsilon^n}{(\epsilon^\alpha)^n}$, where n is the dimensionality of our dataset. We obtain the expected (modeled) number of batches that will be executed as:

$$n_b = \left\lceil V(\epsilon^\alpha, \epsilon) \cdot n_b^\alpha \cdot \left(\frac{b_s^\alpha}{b_s}\right) \right\rceil. \quad (3)$$

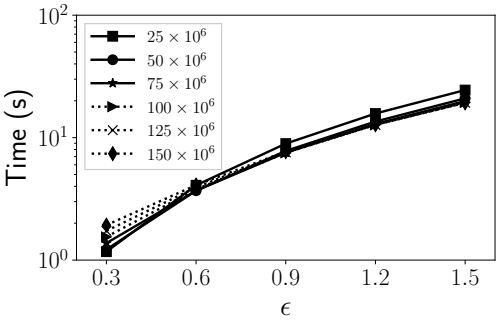
The ceiling is used because we cannot have a fractional number of batches. The modeled value of n_b is used to estimate the total kernel invocation overhead. We model the response time for a given ϵ and b_s as:

$$T^{Model}(\epsilon, b_s, \epsilon^\alpha, n_b, \theta) = T_1(\epsilon^\alpha) \cdot V(\epsilon^\alpha, \epsilon) + \theta \cdot n_b + 3T_2(b_s). \quad (4)$$

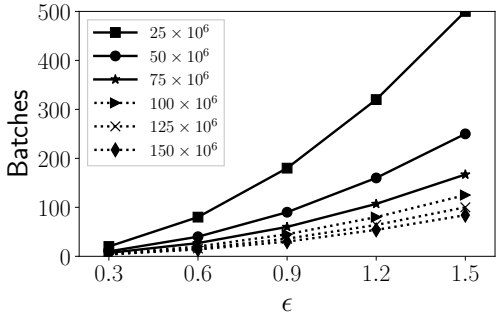
Intuitively, the model simply relies on: (*i*) linearly scaling the time needed to execute GPU-SJ without overheads, $T_1(\epsilon^\alpha)$, by the expected amount of work as a function of the ratio of ϵ search radius volumes, $V(\epsilon^\alpha, \epsilon)$; (*ii*) scaling the expected number of kernel invocations as a function of the above-mentioned ratio; and, (*iii*) computing the time to allocate pinned memory.

5.4.3. Model Validation

We evaluate the accuracy of our performance model using the *SW2DA* dataset with the following parameters. We execute GPU-SJ for $b_s^\alpha = 150 \times 10^6$ and $b_s^\beta = 50 \times 10^6$, and $\epsilon^\alpha = 0.9$ (the median ϵ value in Figure 14). Using a batch size of $b_s^\alpha = 150 \times 10^6$ produces $n_b^\alpha = 30$, and $b_s^\beta = 50 \times 10^6$ produces $n_b^\beta = 89$. Using the response time measured when executing GPU-SJ with b_s^α, we compute $T_2(b_s^\alpha) = 0.4$ s (note that we could also measure T_2 using the result of executing GPU-SJ



(a) Response time vs. ϵ



(b) Number of batches, n_b, vs. ϵ

Figure 16: Performance model results for different batch sizes, b_s, on *SW2DA*. Execution performed on Platform2

with b_s^β). We assume that pinned memory allocation time scales linearly with b_s, so we compute it for a given b_s as $T_2(b_s) = 0.4 \cdot (b_s / b_s^\alpha)$. Thus, using 3 streams, for b_s values of (25, 50, 75, 100, 125, and 150)$\times 10^6$, we compute the pinned memory allocation overhead to be $3T_2(b_s)$ = 0.2 s, 0.4 s, 0.6 s, 0.8 s, 1.0 s, and, 1.2 s, respectively. Using the formulation in Equation 2 we obtain $\theta = 0.015$ s. Finally, with $b_s^\alpha = 150 \times 10^6$, and $n_b^\alpha = 30$, we obtain $T_1(\epsilon^\alpha)$ =6.039 s.

With the two executions of GPU-SJ for $\epsilon^\alpha = 0.9$ (corresponding to the batch sizes b_s^α and b_s^β) we estimate the response time for all values of ϵ and batch sizes, b_s, with results shown in Figure 16 (a). At the smallest and largest workloads ($\epsilon = 0.3$, and $\epsilon = 1.5$, respectively), we find that the relative ordering of the modeled response times for the various batch sizes are consistent with measured results (Figure 14 (a)). This indicates that the model can be used to select a batch size to minimize the response time.

We note that, for $b_s = 25 \times 10^6$ (the smallest batch size) and $\epsilon = 1.5$, there is a large discrepancy between the modeled (Figure 16 (a)) and measured (Figure 14 (a)) response times. This is because, for this small batch size there are many (456) kernel invocations (Figure 14 (b)), each of which is very small ($|D|/456 = 4,089$). This results in too few threads to saturate the GPU's resources. Since the model parameters are computed with larger batch sizes that *do* saturate the GPU's resources, and we scale the response time by the range query volume ratio, the model underestimates the response time. Nevertheless, it is clear from Figure 14 (a) that using $b_s = 25 \times 10^6$ for $\epsilon = 1.5$ yields the worst performance, and this is consistent with the modeled response time. By comparing Figures 14 (a) and 16 (a), we see that the model underestimates the response time when the number of batches exceeds ~ 100. On this dataset, this corresponds to roughly 5× the number of GPU cores, indicating that at least 5 threads per core is needed to saturate the GPU.

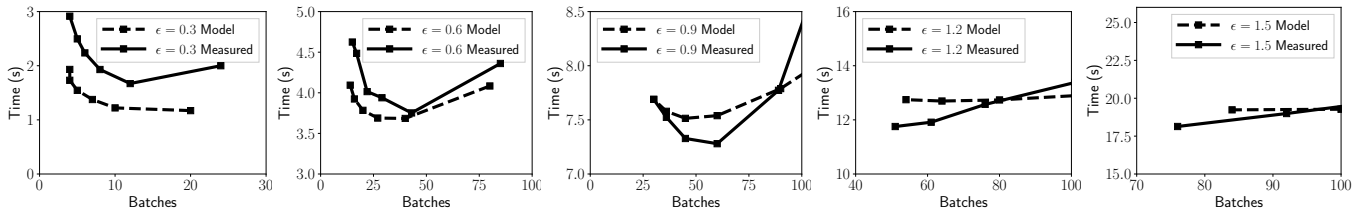
We make a direct comparison between the modeled and measured response times to determine the accuracy of our model and to select a good value of b_s as a function of ϵ. Figure 17 plots the modeled and measured response times vs. the number of batches for each value of ϵ shown in Figure 14. Since the model underestimates response time when $n_b > 100$, we exclude these values in Figure 17. The model can be used to select a good value for b_s to minimize the response time, instead of selecting a single value of b_s (as elected in the experimental evaluation). Table 6 shows the batch size selected by the model (to provide the fastest response time), and the batch size from the experimental results that yields the fastest response time. In 3 of 5 values of ϵ, the model predicts the best batch size that minimizes the response time (the slowdown is shown as 1). In 2 of 5 values, the model incorrectly predicts the best batch size; however, the predicted batch size provides good performance, with a slowdown of at most 0.98× compared to the ideal batch size.

Table 6: Modeled best batch size vs. actual best batch size. Bold face indicates a perfect prediction.

ϵ	Best b_s Model	Best b_s Real	Slowdown
0.3	**25×10⁶**	**25×10⁶**	1
0.6	**50×10⁶**	**50×10⁶**	1
0.9	100×10⁶	75×10⁶	0.993
1.2	125×10⁶	150×10⁶	0.987
1.5	**150×10⁶**	**150×10⁶**	1

As described above, the results presented in Section 5.3 use a fixed value of $b_s = 5.0 \times 10^7$. However, we find that the batch size can significantly impact performance. A small batch size reduces pinned memory overheads, while a large batch size is needed to avoid excessive kernel invocation overhead. On the *SW2DA* dataset using Platform2, we find that, for $\epsilon = 0.3$, using the best batch size (2.5×10^7) results in an average response time of 1.328 s (a speedup of 1.85× over SuperEGO). For $\epsilon = 1.5$, using the best batch size (1.5×10^8) takes 18.136 s (2.01× speedup over SuperEGO). Using the fixed size buffer (Figure 6), however, resulted in a speedup of 1.43× and 1.19×, for $\epsilon = 0.3$ and $\epsilon = 1.5$, respectively. Thus, using our model to select b_s can significantly improve performance.

In summary, we advance a performance model that only requires two executions of GPU-SJ for different batch sizes with a fixed value of ϵ to compute parameters that enable us to estimate the response time for any search radius, ϵ, and batch size, b_s. The performance model relies on a geometric formulation that scales the modeled response time by the difference in range query search volumes, which we assume is a proxy for the total amount of work, and the expected number of batches/kernel in-



(a) $\epsilon = 0.3$ (b) $\epsilon = 0.6$ (c) $\epsilon = 0.9$ (d) $\epsilon = 1.2$ (e) $\epsilon = 1.5$

Figure 17: Comparison of the modeled and measured response time vs. the number of batches, n_b, for each value of ϵ in Figure 14. We limit the results to $n_b \leq 100$. The number of batches correspond to a value of b_s. The larger the result set size, the greater the number of batches, as shown with increasing ϵ.

vocations. The batch size parameter significantly impacts performance, and our model can successfully select a batch size that reduces overall response time. We used a fixed batch size in our performance comparison in Section 5.3; however, further performance gains can be realized by tuning the b_s parameters.

6. Discussion & Conclusion

The self-join is a widely used operation in many data intensive search algorithms and Big Data analytic 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 2.39×, and also significantly outperforms the search-and-refine strategy, CPU-RTREE. Given that the performance of (self-)join algorithms are often sensitive to the input data distribution, we presented results for synthetic datasets of uniformly and skewed data distributions that span 2–6 dimensions, and we evaluated GPU-SJ on 6 real-world datasets.

We show that we can achieve further performance gains by changing the batch size parameter to match the characteristics of the workload. Our performance model shows that the performance of GPU-SJ is sensitive to the pinned memory allocation and kernel invocation overheads. Thus, our model enables us to select a batch size that achieves a trade-off between these overheads.

The high aggregate memory bandwidth and massive parallelism afforded by the GPU makes the architecture attractive for data-intensive algorithms and applications such as the self-join. Additionally, GPUs have low monetary cost per unit performance (e.g., FLOPS). For example, on PLATFORM1, the CPUs are ∼3× more expensive than the Titan X GPU that we used in the performance evaluation. While one should be cautious of hardware cost comparisons, this work demonstrates how the self-join problem can be cost-effectively computed on GPU architectures. Furthermore, GPU-SJ can be executed on multi-GPU systems by simply assigning the batches to different GPUs, further improving performance for a small additional monetary cost relative to the CPUs.

Acknowledgment

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

[1] M. Ester, H. Kriegel, J. Sander, 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, 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, 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, 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, A. Swami, Efficient similarity search in sequence databases, Foundations of data organization and algorithms (1993) 69–84.

[6] C. Böhm, B. Braunmüller, M. Breunig, 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, 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 22 (4) (2013) 561–585.

[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, H. Samet, Metric space similarity joins, ACM Trans. Database Syst. 33 (2) (2008) 7:1–7:38.

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

[12] R. J. Durrant, A. Kabán, When is 'nearest neighbour' meaningful: A converse theorem and implications, Journal of Complexity 25 (4) (2009) 385–397. doi:10.1016/j.jco.2009.02.011.

[13] I. Volnyansky, 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, B. Nam, Exploiting massive parallelism for indexing multi-dimensional datasets on the gpu, IEEE Transactions on Parallel and Distributed Systems 26 (8) (2015) 2258–2271.

[15] Nvidia volta, http://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf, accessed: 2018-4-2.

[16] M. Gowanlock, B. Karsin, GPU Accelerated Self-join for the Distance Similarity Metric, in: IEEE Intl. Workshop on High-Performance Big Data, Deep Learning, and Cloud Computing, in IEEE IPDPS Workshops, accepted for publication, 2018.
URL https://arxiv.org/abs/1803.04120

[17] C. Bohm, H. P. Kriegel, A cost model and index architecture for the sim-

ilarity join, in: Proc. 17th Intl. Conf. on Data Engineering, 2001, pp. 411–420. doi:10.1109/ICDE.2001.914854.
[18] A. Arasu, V. Ganti, R. Kaushik, Efficient exact set-similarity joins, in: Proc. of the Intl. Conf. on Very Large Data Bases, 2006, pp. 918–929.
[19] N. Beckmann, H.-P. Kriegel, R. Schneider, 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. doi:10.1145/93597.98741.
[20] S. Berchtold, D. A. Keim, 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.
[21] J. L. Bentley, Multidimensional binary search trees used for associative searching, Communications of the ACM 18 (9) (1975) 509–517.
[22] R. A. Finkel, J. L. Bentley, Quad trees a data structure for retrieval on composite keys, Acta informatica 4 (1) (1974) 1–9.
[23] C. Böhm, B. Braunmüller, F. Krebs, H.-P. Kriegel, Epsilon grid order: An algorithm for the similarity join on massive high-dimensional data, in: ACM SIGMOD Record, Vol. 30, 2001, pp. 379–388.
[24] K. Yang, B. He, R. Fang, M. Lu, N. Govindaraju, Q. Luo, P. Sander, 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.
[25] J. Zhang, S. You, 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.
[26] M. Gowanlock, 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.
[27] M. Gowanlock, H. Casanova, Distance Threshold Similarity Searches: Efficient Trajectory Indexing on the GPU, IEEE Transactions on Parallel and Distributed Systems 27 (9) (2016) 2533–2545.
[28] J. Kim, B. Nam, Co-processing heterogeneous parallel index for multi-dimensional datasets, Journal of Parallel and Distributed Computing 113 (2018) 195 – 203.
[29] T. D. Han, 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.
[30] M. D. Lieberman, J. Sankaranarayanan, H. Samet, A fast similarity join algorithm using graphics processing units, in: IEEE 24th Intl. Conf. on Data Engineering, 2008, pp. 1111–1120.
[31] M. Gowanlock, C. M. Rude, D. M. Blair, J. D. Li, V. Pankratius, Clustering Throughput Optimization on the GPU, in: Proc. of the IEEE Intl. Parallel and Distributed Processing Symposium, 2017, pp. 832–841.
[32] M. Burtscher, R. Nasre, K. Pingali, A quantitative study of irregular programs on GPUs, in: IEEE Intl. Symposium on Workload Characterization, 2012, pp. 141–151.
[33] P. Guo, L. Wang, P. Chen, A performance modeling and optimization analysis tool for sparse matrix-vector multiplication on gpus, IEEE Transactions on Parallel and Distributed Systems 25 (5) (2014) 1112–1123.
[34] W. Yang, K. Li, Z. Mo, K. Li, Performance optimization using partitioned spmv on gpus and multicore cpus, IEEE Transactions on Computers 64 (9) (2015) 2623–2636.
[35] W. Yang, K. Li, K. Li, A hybrid computing method of spmv on cpu–gpu heterogeneous computing systems, Journal of Parallel and Distributed Computing 104 (2017) 49–60.
[36] M. Gowanlock, C. M. Rude, D. M. Blair, J. Li, V. Pankratius, A Hybrid Approach for Optimizing Parallel Clustering Throughput using the GPU, IEEE Transactions on Parallel and Distributed Systems (2018) 1–1doi:10.1109/TPDS.2018.2869777.
[37] C. Böhm, R. Noll, C. Plant, B. Wackersreuther, A. Zherdin, Data mining using graphics processing units, in: Transactions on Large-Scale Data- and Knowledge-Centered Systems I, Springer, 2009, pp. 63–90.
[38] B. He, K. Yang, R. Fang, M. Lu, N. Govindaraju, Q. Luo, P. Sander, Relational joins on graphics processors, in: Proceedings of the 2008 ACM SIGMOD international conference on Management of data, ACM, 2008, pp. 511–524.
[39] NVIDIA, CUDA programming guide 9.0 (2017).
URL http://docs.nvidia.com/cuda
[40] H. Shamoto, K. Shirahata, A. Drozd, H. Sato, S. Matsuoka, Gpu-accelerated large-scale distributed sorting coping with device memory capacity, IEEE Transactions on Big Data 2 (1) (2016) 57–69. doi:10.1109/TBDATA.2015.2511001.
[41] ftp://gemini.haystack.mit.edu/pub/informatics/dbscandat.zip, accessed 17-September-2017.
[42] 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 219 12.
[43] M. Lichman, UCI machine learning repository (2013).
URL http://archive.ics.uci.edu/ml
[44] I. Kamel, C. Faloutsos, On Packing R-trees, in: Proc. of the Second Intl. Conf. on Information and Knowledge Management, CIKM '93, 1993, pp. 490–499.
[45] B. Karsin, V. Weichert, H. Casanova, J. Iacono, N. Sitchinava, Analysis-driven engineering of comparison-based sorting algorithms on GPUs, in: Proc. of the Intl. Conf. on Supercomputing, ACM, 2018.
[46] NVIDIA, Nsight, http://www.nvidia.com/object/nsight.html (2015).
[47] http://docs.nvidia.com/cuda/pascal-tuning-guide/index.html#memory-throughput, accessed 19-January-2018.

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

Ben Karsin received a Ph.D. in computer science at the University of Hawai'i at Mānoa. As of August 2018, he is a postdoctoral fellow in the Algorithms Research Group at Université libre de Bruxelles. His research interests include parallel algorithms, computational geometry, and GPU architecture.