HEGJoin: Heterogeneous CPU-GPU Epsilon Grids for Accelerated Distance Similarity Join

Benoit Gallet[0000−0001−9716−1502] and Michael Gowanlock[0000−0002−0826−6204]

School of Informatics, Computing, and Cyber Systems, Northern Arizona University, Flagstaff, AZ, 86011, USA
{benoit.gallet, michael.gowanlock}@nau.edu

Abstract. The distance similarity join operation joins two datasets (or tables), A and B, based on a search distance, ϵ, ($A \ltimes_\epsilon B$), and returns the pairs of points (p_a, p_b), where $p_a \in A$ and $p_b \in B$ such that the distance between p_a and $p_b \leq \epsilon$. In the case where $A = B$, then this operation is a similarity self-join (and therefore, $A \bowtie_\epsilon A$). In contrast to the majority of the literature that focuses on either the CPU or the GPU, we propose in this paper Heterogeneous CPU-GPU Epsilon Grids Join (HEGJoin), an efficient algorithm to process a distance similarity join using both the CPU and the GPU. We leverage two state-of-the-art algorithms: LBJoin for the GPU and Super-EGO for the CPU. We achieve good load balancing between architectures by assigning points with larger workloads to the GPU and those with lighter workloads to the CPU through the use of a shared work queue. We examine the performance of our heterogeneous algorithm against LBJoin, as well as Super-EGO by comparing performance to the upper bound throughput. We observe that HEGJoin consistently achieves close to this upper bound.

Keywords: Heterogeneous CPU-GPU Computing · Range Query · Similarity Join · Super-EGO

1 Introduction

Distance similarity searches find objects within a search distance ϵ from a set of query points (or feature vectors). These searches are extensively used in database systems for fast query processing. Due to the high memory bandwidth and computational throughput, GPUs (Graphics Processing Units) have been used to improve database performance (e.g., similarity joins [8], high dimensional similarity searches [19], and indexing methods for range queries [2, 16, 17, 21, 24]). Despite the GPU's attractive performance characteristics, it has not been widely utilized to improve the throughput of modern database systems. Consequently, there has been limited research into concurrently exploiting the CPU and GPU to improve database query throughput. In this paper, we propose a hybrid CPU-GPU algorithm for computing distance similarity searches that combines two highly efficient algorithms designed for each architecture. Note the use of the CUDA terminology throughout this paper.

In database systems, distance similarity searches are typically processed using a join operation. We focus on the distance similarity self-join, defined as performing a distance similarity search around each object in a table ($A \bowtie_\epsilon A$). The method employed in this paper could be used to join two different tables using a semi-join operation ($A \ltimes_\epsilon B$), where A is a set of query points, and B is a set of entries in the index. Throughout this paper, while we refer to the self-join, we do not explore the optimizations applicable only to the self-join, and so leave the possibility to semi-join instead.

The complexity of a brute force similarity search is $O(|D|^2)$, where D is the dataset/table. Thus, indexing methods have been used to prune the search to reduce its complexity. In this case, the *search-and-refine* strategy is used, where the *search* of an indexing structure generates a set of candidate points that are likely to be within ϵ of a query point, and the *refine* step reduces the candidate set to the final set of objects within ϵ of the query point using distance calculations.

The distance similarity search literature typically focuses on either low [7, 8, 11, 15] or high [19] dimensional searches. Algorithms designed for low dimensionality are typically not designed for high dimensionality (and vice versa). The predominant reason for this is due to the *curse of dimensionality* [5, 15], where index searches become more exhaustive and thus tend to degrade into a brute force search as the dimensionality increases. In this work, we focus on low-dimensional exact searches that do not dramatically suffer from the curse of dimensionality. Since the cost of the distance calculation used to refine the candidate set increases with dimensionality, low-dimensional searches are often memory-bound, as opposed to compute-bound in high-dimensions. The memory-bound nature of the algorithm in low-dimensionality creates several challenges that may hinder performance and limit the scalability of parallel approaches.

As discussed above, there is a lack of heterogeneous CPU-GPU support for database systems in the literature. This paper proposes an efficient distance similarity search algorithm that uses both the CPU and the GPU. There are two major CPU-GPU similarity search algorithm designs, described as follows:

•**Task parallelism:** Assign the CPU and GPU particular tasks to compute. For example, Kim and Nam [18] compute range queries using the CPU to search an R-Tree while the GPU refines the candidate set, while Shahvarani and Jacobsen [23] use the CPU and the GPU to search a B-Tree, and then use the CPU to refine the candidate set.

•**Data parallelism:** Split the work and perform both the search and refine steps on each architecture independently, using an algorithm suited to each architecture. To our knowledge, while no other works have used the data-parallel approach in a hybrid CPU-GPU distance similarity search algorithm, we found that this design has been used by Gowanlock [12] for k Nearest Neighbor (kNN) searches. In this paper, we focus on the *data-parallel* approach because it allows us to assign work to each architecture based on the workload of each query point. In modern database systems, this approach allows us to exploit all available computational resources in the system, which maximizes query throughput.

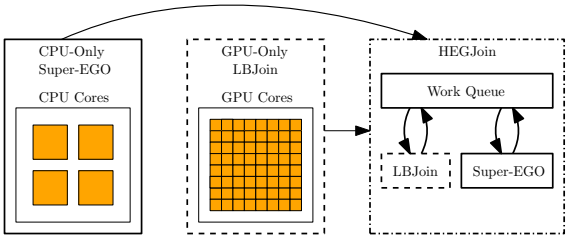


Fig. 1. Representation of how we combine SUPER-EGO and LBJOIN by using a single work queue to form HEGJOIN.

Our algorithm leverages two previously proposed independent works that were shown to be highly efficient: the GPU algorithm (LBJOIN) by Gallet and Gowanlock [11] and the CPU algorithm (SUPER-EGO) presented by Kalashnikov [15]. Figure 1 illustrates how these algorithms work together through the use of a single shared work queue. By combining these two algorithms, we achieve better performance on most experimental scenarios than CPU-only or GPU-only approaches. This paper makes the following contributions:

•We combine state-of-the-art algorithms for the CPU and GPU (Section 3).

•We propose an efficient double-ended work queue (deque) that assigns work on-demand to the CPU and GPU algorithms. This allows both architectures to be saturated with work while achieving low load imbalance (Section 4).

•By using the work queue, we split the work between the CPU and GPU by allowing the CPU (GPU) to compute the query points with the lowest (highest) workload. This exploits the GPU's high computational throughput.

•We optimize SUPER-EGO to further improve the performance of our hybrid algorithm. We denote this optimized version of SUPER-EGO as SEGO-NEW.

•We evaluate the performance using five real-world and ten exponentially distributed synthetic datasets. We achieve speedups up to 2.5× (11.3×) over the GPU-only (CPU-only) algorithms (Section 5).

The rest of the paper is organized as follows: we present background material in Section 2, and conclude the paper in Section 6.

2 Background

2.1 Problem Statement

Let D be a dataset in d dimensions. Each point in D is denoted as q_i, where $i = 1, ..., |D|$. We denote the j^{th} coordinate of $q_i \in D$ as $q_i(j)$, where $j = 1, ..., d$. Thus, given a distance threshold ϵ, we define the distance similarity search of a query point q as finding all points in D that are within this distance ϵ to q. We also define a candidate point $c \in D$ as a point whose distance to q is evaluated. Similarly to related work, we use the Euclidean distance. Therefore, the similarity join finds pairs of points $(q \in D, c \in D)$, such that $dist(q, c) \leq \epsilon$, where $dist(q, c) = \sqrt{\sum_{j=1}^{d}(q(j) - c(j))^2}$. All processing occurs in-memory. While

we consider the case where the result set size may exceed the GPU's global memory capacity, we do not consider the case where the result set size may exceed the platform's main memory capacity.

2.2 Related Work

We present relevant works regarding the distance similarity join. Since the similarity join is frequently used as a building block within other algorithms, the literature regarding the optimization of the similarity join is extensive. However, the vast majority of existing literature aims at improving performance using either the CPU or the GPU, and rarely both. Hence, literature regarding heterogeneous CPU-GPU similarity join optimizations remains relatively scarce. The search-and-refine strategy (Section 1) largely relies on the use of data indexing methods that we describe as follows.

Indexes are used to prune searches. Given a query point q and a distance threshold ϵ, indexes find the candidate points that are likely to be within a distance ϵ of q. Also, the majority of the indexes are designed for a specific use, whether they are for low or high dimensional data, for the CPU, for the GPU or for both architectures. We identify different indexing methods, including those designed for the CPU [3, 4, 6, 7, 9, 14, 15, 22], the GPU [2, 13, 17], or both architectures [12, 18, 23]. As our algorithm focuses on the low dimensionality distance similarity search, we focus on presenting indexing methods also designed for lower dimensions. Since indexes are an essential component of distance similarity searches, identifying the best index for each architecture is critical to achieve good performance, especially when using two different architectures. Furthermore, although our heterogeneous algorithm leverages two previously proposed works [11,15] that both use a grid indexing for the CPU and the GPU, we discuss in the following sections several other indexing methods based on trees.

CPU Indexing: In the literature, the majority of indexes designed for the CPU to index multi-dimensional data are based on trees, such as the kD-Tree [6], the Quad Tree [10] or the R-Tree [14]. The B-Tree [3] is designed to index 1-dimensional data. All these indexes are designed for range queries and can be used for distance similarity searches. Grid indexes such as the Epsilon Grid Order (EGO) [7,15] have been designed for distance similarity joins. We discuss this EGO index that we leverage in Section 3.2.

GPU Indexing: Similarly to CPU indexes, index-trees have been specially optimized to be efficient on the GPU. The R-Tree has been optimized by Kim and Nam [17] and the B-Tree by Awad et al. [2], both designed for range queries. As an example of the optimizations they make to the R-Tree and the B-Tree, both works mostly focus on removing recursive accesses inherent to tree traversals or on reducing threads divergence. We present the grid index proposed by Gowanlock and Karsin [13] designed for distance similarity joins and that we leverage in Section 3.1.

CPU-GPU Indexing: Kim and Nam [18] propose an R-Tree designed for range queries that uses task parallelism. The CPU searches the internal nodes of the tree, while the GPU refines the objects in the leaves. Gowanlock [12] instead

elects to use two indexes for data parallelism to compute kNN searches: the CPU uses a kD-Tree while the GPU uses a grid, so both indexes are suited to their respective architecture.

3 Leveraged Work

In this section, we present the previously proposed works we leverage to design HEGJoin. Therefore, we use LBJoin [11] for the GPU and Super-EGO [15] for the CPU, two state-of-the-art algorithms on their respective platforms. The GPU[1] and CPU[2] algorithms are publicly available.

3.1 GPU Algorithm: LBJoin

The GPU component of HEGJoin is based on the GPU kernel proposed by Gallet and Gowanlock [11]. This kernel also uses the grid index and the batching scheme by Gowanlock and Karsin [13]. This work is the best distance similarity join algorithm for low dimensions that uses the GPU (there are similar GPU algorithms but they are designed for range queries, see Section 2).

Grid Indexing: The grid index presented by Gowanlock and Karsin [13] allows the query points to only search for candidate points within its 3^d adjacent cells (and the query points' own cell), where d is the data dimensionality. This grid is stored in several arrays: *(i)* the first array represents only the non-empty cells to minimize memory usage, *(ii)* the second array stores the cells' linear id and a minimum and maximum indices of the points, *(iii)* the third array corresponds to the position of the points in the dataset and is pointed to by the second array. Furthermore, the threads within the same warp access neighboring cells in the same lock-step fashion, thus avoiding thread divergence. Also, note that we modify their work and now construct the index directly on the GPU, which is much faster than constructing it on the CPU as in the original work.

Batching Scheme: Computing the ϵ-neighborhood of many query points may yield a very large result set and exceed the GPU's global memory capacity. Therefore, in Gowanlock and Karsin [13], the total execution is split into multiple batches, such that the result set does not exceed global memory capacity.

The number of batches that are executed, n_b, are defined by an estimate of the total result set size, n_e, and a buffer of size n_s, which is stored on the GPU. The authors use a lightweight kernel to compute n_e, based on a sample of D. Thus, they compute $n_b = n_e/n_s$.[3] The buffer size, n_s, can be selected such that the GPU's global memory capacity is not exceeded. The number of queries, n_q^{GPU}, processed per batch (a fraction of $|D|$) are defined by the number of batches as follows: $n_q^{GPU} = |D|/n_b$. Hence, a smaller number of batches will yield a larger number of queries processed per batch.

[1] https://github.com/benoitgallet/self-join-hpbdc2019, last accessed: Feb. 27^{th}, 2020

[2] https://www.ics.uci.edu/~dvk/code/SuperEGO.html, last accessed: Feb. 27^{th}, 2020

[3] In this section, for clarity, and without the loss of generality, we describe the batching scheme assuming all values divide evenly.

The total result set is simply the union of the results from each batch. Let R denote the total result set, where $R = \bigcup_{l=1}^{n_b} r_l$, where r_l is the result set of a batch, and where $l = 1, 2, \ldots, n_b$.

The batches are executed in three CUDA *streams*, allowing the overlap of GPU computation and CPU-GPU communication, and other host-side tasks (e.g., memory copies into and out of buffers), which is beneficial for performance.

Sort by Workload and Work Queue: The sorting strategy proposed by Gallet and Gowanlock [11] sorts the query points by non-increasing workload. The workload of a query point is determined by the sum of candidate points in its own cell and its 3^d adjacent cells in the grid index. This results in a list of query points sorted from most to least workload, which is then used in the work queue to assign work to the GPU's threads. The consequence of sorting by workload and of using this work queue is that threads within the same warp will compute query points with a similar workload, thereby reducing intra-warp load imbalance. This reduction in load imbalance, compared to their GPU reference implementation [11], therefore reduces the overall number of periods where some threads of the warp are idle and some are computing. This yields an overall better response time than when not sorting by workload. This queue is stored on the GPU as an array, and a variable is used to indicate the head of the queue. In this paper, we store this queue on the CPU's main memory to be able to share the work between the CPU and the GPU components of HEGJoin.

GPU Kernel: The GPU kernel [11] makes use of a grid index, the batching scheme, as well as the sorting by workload strategy and the work queue presented above. Moreover, we configure the kernel [11] to use a single GPU thread to process each query point ($|D|$ threads in total). Thus, each thread first retrieves a query point from the work queue using an atomic operation. Then, using the grid index, the threads search for their non-empty neighboring cells corresponding to their query point, and iterate over the found cells. Finally, for each point within these cells, the algorithm computes the distance to the query point and if this distance is $\leq \epsilon$, then the key/value pair made of the query point's id and the candidate point's id is added to the result buffer r of the batch.

3.2 CPU Algorithm: Super-EGO

Similarly to our GPU component, the CPU component of our heterogeneous algorithm is based on the efficient distance similarity join algorithm, Super-EGO, proposed by Kalashnikov [15]. We present its main features as follows.

Dimension Reordering: The principle of this technique is to first compute a histogram of the average distance between the points of the dataset and for each dimension. A dimension with a high average distance between the points means that points are more spread across the search space, and therefore fewer points will join. The goal is to quickly increase the cumulative distance between two points so it reaches ϵ with fewer distance calculations, allowing the algorithm to short-circuit the distance calculation and continue computing the next point.

EGO Sort: This sorting strategy sorts the points based on their coordinates in each dimension, divided by ϵ. This puts spatially close points close to each other in memory, and serves as an index to find candidate points when joining two sets of points. This sort was originally introduced by Böhm et al. [7].
Join Method: The SUPER-EGO algorithm joins two partitions of the datasets together, and recursively creates new partitions until it reaches a given size. Then, since the points are sorted based on their coordinates and the dimensions have been reordered, two partitions are compared only if their first point is within ϵ from each other. If they are not, then subsequent points will not join either and the join of the two partitions is aborted. The join is thus made directly on several partitions of the input datasets.
Parallel Algorithm: SUPER-EGO also adds parallelism to the original EGO algorithm, using PTHREADS and a producer-consumer scheme to balance the workload between threads. When a new partition is recursively created, if the size of the queue is less than the number of threads (i.e., some threads have no work), the newly created partitions are added to the work queue to be shared among the threads. This ensures that no threads are left without work to compute.

4 Heterogeneous CPU-GPU Algorithm: HEGJoin

We present the major components of our heterogeneous CPU-GPU algorithm, HEGJOIN, as well as improvements made to the leveraged work (Section 3).

4.1 Shared Work Queue

As mentioned in Section 3.1, we reuse the work queue stored on the GPU that was proposed by Gallet and Gowanlock [11], which efficiently balances the workload between GPU threads. However, to use the work queue for the CPU and the GPU components, we must relocate it to the host/CPU to use it with our CPU algorithm component. Because the GPU has a higher computational throughput than the CPU, we assign the query points with the most work to the GPU, and those with the least work to the CPU. Similarly to the shared work queue proposed by Gowanlock [12] for the CPU-GPU kNN algorithm, the query points need to be sorted based on their workload, as detailed in Section 3.1. However, while query points' workload in Gowanlock [12] is characterized by the number of points within each query point's cell, we define here the workload as the number of candidate points within all adjacent cells. Our sorting strategy is more representative of the workload than in Gowanlock [12], as it yields the exact number of candidates that must be filtered for each query point.

Using this queue with the CPU and the GPU requires modifying the original work queue [11] to be a double ended-queue (deque), as well as defining a deque index for each architecture. Since the query points are sorted by workload, we set the GPU's deque index to the beginning of the deque (greatest workload), and to the end of the deque for the CPU's index (smallest workload). Therefore, the GPU's workload is configured to decrease while CPU's workload increases,

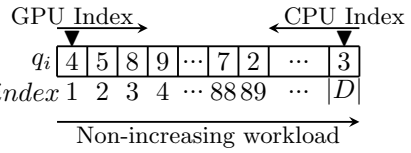


Fig. 2. Representation of our deque as an array. The numbers q_i are the query points id, the triangles are the starting position of each index, and the arrows above it indicate the indices progression in the deque.

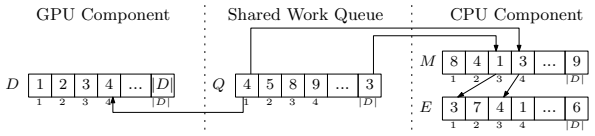


Fig. 3. Illustration of an input dataset D, the shared deque sorted by workload Q, the input dataset EGO-sorted E and the mapping M between Q and E. The numbers in D, Q, and E correspond to query point ids, while the numbers in M correspond to their position in E. The numbers below the arrays are the indices of the elements.

as their respective index progresses in the deque. Also, note that while n_q for the CPU (n_q^{CPU}) is fixed, n_q for the GPU (n_q^{GPU}) varies based on the dataset characteristics and on ϵ (Section 3.1).

We assign query points from the deque to each architecture, described as follows: (a) We set the GPU's deque index to 1 and the CPU's deque index to $|D|$; (b) We create an empty batch if the GPU's index and the CPU's index are at the same position in the deque, and the program terminates; (c) To assign query points to the GPU, we create and assign a new batch to the GPU, and increase GPU's deque index; (d) To assign query points to the CPU, we create and assign a new batch to the CPU, and decrease CPU's deque index.

As described in Section 3, HEGJoin uses two different sorts: sorting by workload (Section 3.1) and Super-EGO's EGO-sort (Section 3.2). However, as these two strategies sort following different criteria, it is not possible to first sort by workload then to EGO-sort (and vice-versa), as the first sort would be overwritten by the second sort. We thus create a mapping between the EGO-sorted dataset and our shared work queue, as represented in Figure 3.

4.2 Batching Scheme: Complying with Non-Increasing Workload

A substantial issue arises when combining the batching scheme and the sorting by workload strategy (Section 3.1). As the batch estimator creates batches with a fixed number of query points, and because the query points are sorted by workload, the original batching scheme creates successive batches with a non-increasing workload. Hence, as the execution proceeds, the batches take less time to compute and the overhead of launching many kernels may become substantial, especially as the computation could have been executed with fewer batches.

We modify the batching scheme (Section 3.1) to accommodate the sorting by workload strategy, and that we represent in Figure 4. While still estimating a fraction of the points, the rest of the points get a number of neighbors inferred from the maximum value of the two closest estimated points (to overestimate

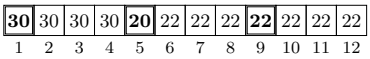


Fig. 4. Representation of the new batch estimator. The bold numbers are the estimated number of neighbors of those points, while the other numbers are inferred, based on the maximum result between the two closest estimated points shown in bold.

and avoid buffer overflow during computation). Adding the estimated and the inferred numbers of neighbors yields an estimated result set size n_e. We then create the batches so they have a consistent result set size r_l close to the buffer size n_s. As the number of estimated neighbors should decrease (as their workload decreases), the number of query points per batch increases. Furthermore, we set a minimum number of batches to $2 \times n^{streams}$, where $n^{streams} = 3$ is the number of CUDA streams used. Therefore, the GPU can only initially be assigned up to half of the queries in the work queue. This ensures that the GPU is not initially assigned too many queries, which would otherwise starve the CPU of work to compute.

4.3 GPU Component: HEGJoin-GPU

The GPU component of our heterogeneous algorithm, which we denote as HEGJoin-GPU and that we can divide into two parts (the host and the kernel), remains mostly unchanged from the algorithm proposed by Gallet and Gowanlock [11] and presented in Section 3.

Regarding the host side of our GPU component, we modify how the kernels are instantiated to use the shared work queue presented in Section 4.1. Therefore, as the original algorithm was looping over all the batches (as given by the batch estimator, presented in Section 3), we loop while the shared deque returns a valid batch to execute (Section 4.1).

In the kernel, since the work queue has been moved to the CPU, a batch corresponds to a range of queries in the deque whose interval is determined when taking a new batch from the queue, and can be viewed as a "local queue" on the GPU. Therefore, the threads in the kernel update a counter local to the batch to determine which query point to compute, still following the non-increasing workload that yields a good load balancing between threads in the same warp.

4.4 CPU Component: HEGJoin-CPU

The CPU component of HEGJoin is based on the Super-EGO algorithm proposed in [15] and presented in Section 3.2. We make several modifications to Super-EGO to incorporate the double ended queue we use, and we also optimize Super-EGO to improve its performance.

As described in Section 3.2, Super-EGO uses a queue and a producer-consumer system for multithreading. We remove this system and replace it with our shared double-ended queue. Because the threads are continuously taking work from the shared deque until it is empty, the producer-consumer originally used becomes unnecessary, as the deque signals Super-EGO when it is empty.

Table 1. Summary of the real-world datasets used for the experimental evaluation. $|D|$ denotes the number of points and d the dimensionality.

Dataset	$\|D\|$	d	Dataset	$\|D\|$	d	Dataset	$\|D\|$	d
SW2DA	1.86M	2	*SW2DB*	5.16M	2	*SDSS*	15.23M	2
SW3DA	1.86M	3	*SW3DB*	5.16M	3			

The original SUPER-EGO algorithm recursively creates sub-partitions of contiguous points on the input datasets until their size is suited for joining. As one of the partitions is now taken from our deque, which is sorted by workload, it no longer corresponds to a contiguous partition of the input dataset. Thus, we loop over the query points of the batch given by the deque to join it with the other points in the partition. This optimization requires the use of the mapping presented in Section 4.1 and illustrated in Figure 3.

SUPER-EGO uses QSORT from the C standard library to EGO-sort, and we replace it by the more efficient and parallel BOOST::SORT::SAMPLE_SORT algorithm, a stable sort from the Boost C++ library. This allows SEGO-NEW to start its computation earlier than SUPER-EGO would, as it is faster than QSORT. We use as many threads to sort as we use to compute the join.

Finally, in contrast to the original SUPER-EGO algorithm, this new version of SUPER-EGO is now capable of using 64-bit floats instead of only 32-bit floats.

5 Experimental Evaluation

5.1 Datasets

We use real-world and exponentially distributed synthetic datasets (using $\lambda = 40$), spanning 2 to 8 dimensions. The real-world datasets we select are the *Space Weather* datasets (*SW-*) [20], composed of 1.86M or 5.16M points in two dimensions representing the latitude and longitude of objects, and adding the total number of electrons as the third dimension. We also use data from the *Sloan Digital Sky Survey* dataset (*SDSS*) [1], composed of a sample of 15.23M galaxies in 2 dimensions. A summary of the real-world datasets is given in Table 1.

Additionally, we use synthetic datasets made of 2M and 10M points spanning two to eight dimensions. These datasets are named using the dimensions and number of points: *Expo3D2M* is a 3-dimensional dataset of 2M points. We elect to use an exponential distribution as this distribution contains over-dense and under-dense regions, which are representative of the real-world datasets we select (Table 1). Finally, exponential distributions yield high load imbalance between the points, and should thus be more suited to outline a load imbalance between the processors, an important aspect of HEGJOIN, than uniform distributions.

5.2 Methodology

The platform we use to run our experiments is composed of 2 × Intel Xeon E5-2620v4 with 16 total cores, 128 GiB of RAM, equipped with an Nvidia Quadro

GP100 with 16 GiB of global memory. The code executed by the CPU is written in C/C++, while the GPU code is written using CUDA. We use the GNU compiler and use the O3 optimization flag for all experiments.

We summarize the different implementations we test as follows. For clarity, we differentiate between similar algorithm components since they may use slightly different experimental configurations. For example, we make the distinction between the CPU component of HEGJoin (HEGJoin-CPU) and the original Super-EGO algorithm due to slight variations in their configurations.

LBJoin is the GPU algorithm proposed by Gallet and Gowanlock [11], uses 3 GPU streams (managed by 3 CPU threads), 256 threads per block, $n_s = 5 \times 10^7$ key/value pairs, 64-bit floats, and n_q^{GPU} is given by the batch estimator presented in Section 4.2. **Super-EGO** is the CPU algorithm developed by Kalashnikov [15], uses 16 CPU threads (on 16 physical cores) and 32-bit floats. **SEGO-New** is our optimized version of Super-EGO as presented in Section 4.4, using 16 CPU threads, 64-bit floats and the sorting by workload strategy. **HEGJoin-GPU** is the GPU component of HEGJoin, using the same configuration as LBJoin and our shared work queue (Section 4.1). **HEGJoin-CPU** is the CPU component of HEGJoin, using the same configuration as SEGO-New and our shared work queue, with $n_q^{CPU} = 1,024$ (Section 4.1). Finally, **HEGJoin** is our heterogeneous algorithm using the shared work queue (Section 4.1) to combine HEGJoin-CPU and HEGJoin-GPU.
LBJoin, Super-EGO, SEGO-New and HEGJoin are standalone, and thus compute all the work. HEGJoin-CPU and HEGJoin-GPU, as part of our HEGJoin algorithm, each compute a fraction of the work.

The response times presented are averaged over three trials, and include the end-to-end computation time, i.e., the time to construct the grid index on the GPU, sort by workload, reorder the dimensions and to EGO-sort, and the time to join. Note that some of these time components may overlap (e.g., EGO-sort and GPU computation may occur concurrently).

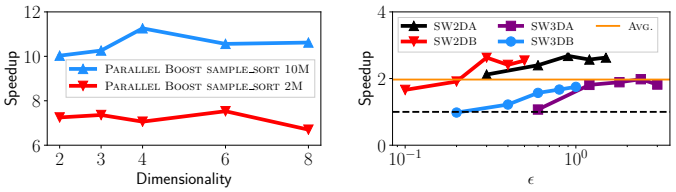
5.3 Selectivity

We report the selectivity (defined by Kalashnikov [15]) of our experiments as a function of ϵ. We define the selectivity $S = (|R| - |D|)/|D|$, where R is the result set and D is the input dataset. Across all experiments, selectivity ranges from 0 on the *Expo8D2M* and *Expo8D10M* datasets to $13,207$ on the *SW3DA* dataset. We include the selectivity when we compare the performance of all algorithms.

5.4 Results

Performance of SEGO-New: We evaluate the optimized version of Super-EGO and denoted as SEGO-New. The major optimizations include a different sorting algorithm, sorting by workload strategy and work queue (Section 4.4).

We evaluate the performance of EGO-sort using the parallel sample_sort algorithm from the C++ Boost library over the qsort algorithm from the C standard library. sample_sort is used by SEGO-New (and thus by HEGJoin),



(a) EGO-Sort Speedup (b) SEGO-New Speedup

Fig. 5. (a) Speedup to EGO-Sort our synthetic datasets using SAMPLE_SORT from the Boost library over QSORT from the C standard library. $S = 0$–9.39K and $S = 0$–1.99K on the 2M and 10M points datasets, respectively. (b) Speedup of SEGO-New over Super-EGO on the *SW-* real-world datasets.

while QSORT is used by Super-EGO. Figure 5(a) plots the speedup of SAMPLE_SORT over QSORT on our synthetic datasets. We observe an average speedup of 7.18× and 10.55× on the 2M and 10M points datasets, respectively. Note that we elect to use the SAMPLE_SORT as the EGO-sort needs to be stable.

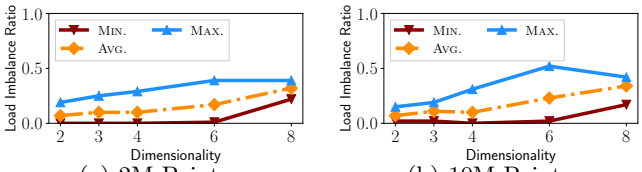
Figure 5(b) plots the speedup of SEGO-New over Super-EGO on the *SW-* real-world datasets. SEGO-New achieves an average speedup of 1.97× over Super-EGO. While SEGO-New uses 64-bit floats, Super-EGO only uses 32-bit floats and is thus advantaged compared to SEGO-New. We explain this overall speedup by using SAMPLE_SORT over QSORT, and the sorting by workload strategy with the work queue. Therefore, SEGO-New largely benefits from balancing the workload between its threads and from using the work queue.

Evaluating the Load Balancing of the Shared Work Queue: In this section, we evaluate the load balancing efficiency of our shared work queue, which can be characterized as the time difference between the CPU and the GPU ending their respective work. Indeed, a time difference close to 0 indicates that both CPU and GPU components of HEGJoin ended their work at a similar time, and therefore that their workload was balanced.

Figure 6 plots the ratio of load imbalance as the time difference between the CPU (HEGJoin-CPU) and the GPU (HEGJoin-GPU) ending their respective work over the total response time of the application on our exponentially distributed synthetic datasets spanning 2 to 8 dimensions with (a) 2M and (b) 10M points. While we can observe a relatively high maximum load imbalance, these cases arise when the selectivity, and so the workload, are low. As the workload increases our deque becomes more efficient, and the load imbalance is reduced.

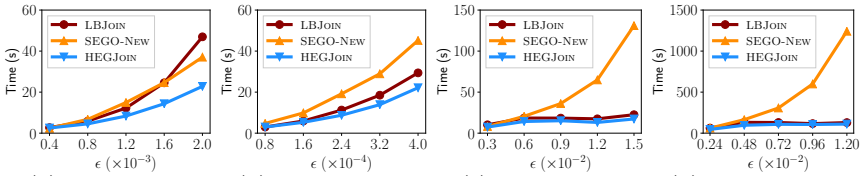
Performance of HEGJoin: We now compare the overall response time vs. ϵ of HEGJoin, LBJoin, and SEGO-New. Note that we decide to use SEGO-New instead of Super-EGO as it performs consistently better (Figure 5(b)).

Figure 7 plots the response time vs. ϵ of HEGJoin, LBJoin and SEGO-New on (a) *Expo2D2M*, (b) *Expo2D10M*, (c) *Expo8D2M* and (d) *Expo8D10M*. We select these datasets as they constitute the minimum and maximum in terms of size and dimensionality among our synthetic datasets, and we observe similar results on the intermediate datasets of different dimensionality and size. Hence,



(a) 2M Points (b) 10M Points

Fig. 6. Load imbalance between HEGJoin-CPU and HEGJoin-GPU using HEGJoin across our synthetic datasets spanning $d = 2$–8. $S = 157$–9.39K on the 2M points datasets (a), and $S = 167$ – 1.99K on the 10M points datasets (b).



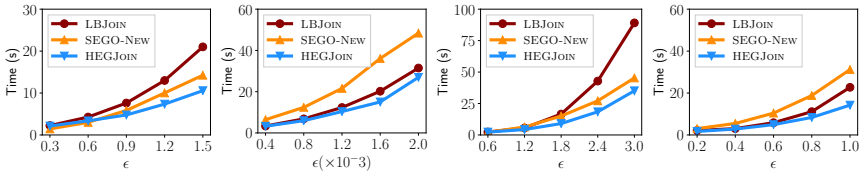
(a) Expo2D2M (b) Expo2D10M (c) Expo8D2M (d) Expo8D10M

Fig. 7. Response time vs. ϵ of LBJoin, SEGO-New and HEGJoin on 2M and 10M point synthetic datasets in 2-D and 8-D. S is in the range (a) 397–9.39K, (b) 80–1.99K, (c) 0–157 and (d) 0–167.

on such datasets, while SEGO-New performs relatively well on the *Expo2D2M* dataset (Figure 7(a)) and even better than LBJoin when $1.6 < \epsilon$, HEGJoin performs better than LBJoin and SEGO-New in all subfigures (Figure 7(a)–(d)). However, as dimensionality increases and as the performance of SEGO-New decreases in 8 dimensions (Figures 7(c) and (d)), combining it with LBJoin does not significantly improve performance. Thus, as SEGO-New (and therefore as HEGJoin-CPU as well) do not scale well with dimensionality, the performance of HEGJoin relies nearly exclusively on the performance of its GPU component HEGJoin-GPU. HEGJoin achieves a speedup of up to 2.1× over only using LBJoin on the *Expo2D2M* dataset (Figure 7(a)), and up to 11.3× over SEGO-New on the *Expo8D10M* dataset (Figure 7(d)).

Figure 8 plots the response time vs. ϵ of LBJoin, SEGO-New and HEGJoin on our real-world datasets. Similarly to Figure 7, as these datasets span 2 and 3 dimensions, the performance of SEGO-New is better than LBJoin on *SW2DA* and *SW3DA* (Figures 8(a) and (c), respectively). Thus, using HEGJoin always improves performance over only using LBJoin or SEGO-New. We achieve a speedup of up to 2.5× over using LBJoin on the *SW3DA* dataset (Figure 8(c)), and up to 2.4× over SEGO-New on the *SDSS* dataset (Figure 8(b)).

From Figures 7 and 8, we observe that using HEGJoin over LBJoin or SEGO-New is beneficial. Typically, in the worst case, HEGJoin performs similarly to the best of LBJoin and SEGO-New, and consistently performs better than using just LBJoin or SEGO-New. Thus, there is no disadvantage to using HEGJoin instead of LBJoin or SEGO-New.



(a) SW2DA (b) SDSS (c) SW3DA (d) SW3DB

Fig. 8. Response time vs. ϵ of LBJoin, SEGO-New and HEGJoin on the (a) *SW2DA*, (b) *SDSS*, (c) *SW3DA* and (d) *SW3DB* real-world datasets. S is in the range (a) 295–5.82K, (b) 1–31, (c) 239–13.20K and (d) 33–2.13K.

Table 2. Query throughput (queries/s) of LBJoin, SEGO-New, the upper bound of LBJoin plus SEGO-New, HEGJoin, and the performance ratio between HEGJoin and the upper bound across several datasets.

Dataset	ϵ	S	LBJoin	SEGO-New	Upper Bound	HEGJoin	Perf. Ratio
Expo2D2M	2.0×10^{-3}	9,392	42,589	53,996	96,585	87,681	0.91
Expo4D2M	1.0×10^{-2}	9,262	14,745	13,557	28,302	26,382	0.93
Expo8D2M	1.5×10^{-2}	157	88,968	15,264	104,232	115,674	1.11
Expo2D10M	4.0×10^{-4}	1,985	340,252	221,288	561,540	451,875	0.80
Expo4D10M	4.0×10^{-3}	1,630	142,816	49,704	192,520	217,297	1.13
Expo8D10M	1.2×10^{-2}	167	77,507	8,055	85,562	90,654	1.06
SW2DA	1.5×10^{0}	5,818	88,749	130,942	219,691	176,574	0.80
SDSS	2.0×10^{-3}	31	485,508	314,771	798,834	567,086	0.71
SW3DA	3.0×10^{0}	13,207	20,930	41,143	62,073	53,093	0.86

Table 2 presents the query throughput for LBJoin, SEGO-New, HEGJoin, as well as an upper bound (the addition of LBJoin and SEGO-New respective throughput), and the ratio of the throughput HEGJoin achieves compared to this perfect throughput. The query throughput corresponds to the size of the dataset divided by the response time of the algorithm, as shown in Figures 7 and 8. We observe a high performance ratio, demonstrating that we almost reach a performance upper bound. Moreover, we also observe that on the *Expo4D10M* and the 8-D datasets, we achieve a ratio of more than 1. We explain this by the fact that LBJoin's throughput includes query points with a very low workload, thus increasing its overall throughput compared to what HEGJoin-GPU achieves. Similarly, SEGO-New's throughput includes query points with a very large workload, thus reducing its overall throughput compared to what HEGJoin-CPU achieves. When combining the two algorithms, we have the GPU computing the query points with the largest workload and the CPU the points with the smallest workload. The respective throughput of each component should, therefore, be lower for the GPU and higher for the CPU, than their throughput when computing the entire dataset. Moreover, performance ratios lower than 1 indicate that there are several bottlenecks, including memory bandwidth limitations, with the peak bandwidth potentially reached when storing the results from the CPU and the GPU. We particularly observe this on low dimensionality and for low selectivity, as it yields less computation

and a higher memory pressure than in higher dimensions or for higher selectivity (Figures 7 and 8). We confirm this by examining the ratio of kernel execution time over the time to compute all batches, using only the GPU. Focusing on the datasets with the minimum and maximum performance ratio from Table 2, we find that *SDSS* has a kernel execution time ratio of 0.16, while *Expo4D10M* a kernel execution time ratio of 0.72. Hence, most of the *SDSS* execution time is spent on memory operations, while *Expo4D10M* execution time is mostly spent on computation. When executing HEGJoin on *SDSS* (and other datasets with low ratios in Table 2), we observe that the use of the GPU hinders the CPU by using a significant fraction of the total available memory bandwidth.

6 Conclusion

The distance similarity join transitions from memory- to compute-bound as dimensionality increases. Therefore, the GPU's high computational throughput and memory bandwidth make the architecture effective at distance similarity searches. The algorithms used in HEGJoin for the CPU and GPU have their own performance advantages: SEGO-New (LBJoin) performs better on lower (higher) dimensions. By combining these algorithms, we exploit more computational resources, and each algorithm's inherent performance niches.

To enable these algorithms to efficiently compute the self-join, we use a double-ended queue that distributes and balances the work between the CPU and GPU. We find that HEGJoin achieves respectable performance gains over the CPU- and GPU-only algorithms, as HEGJoin typically achieves close to the upper bound query throughput. Thus, studying other hybrid CPU-GPU algorithms for advanced database systems is a compelling future research direction.

Acknowledgement

This material is based upon work supported by the National Science Foundation under Grant No. 1849559.

References

1. Alam, S., Albareti, F., Prieto, C., 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** (2015)
2. Awad, M.A., Ashkiani, S., Johnson, R., Farach-Colton, M., Owens, J.D.: Engineering a High-performance GPU B-Tree. In: Proc. of the 24th Symp. on Principles and Practice of Parallel Programming. pp. 145–157 (2019)
3. Bayer, R., McCreight, E.M.: Organization and maintenance of large ordered indexes. Acta Informatica **1**(3), 173–189 (1972)
4. Beckmann, N., Kriegel, H.P., Schneider, R., Seeger, B.: The R*-tree: An Efficient and Robust Access Method for Points and Rectangles. In: Proc. of the ACM SIGMOD Intl. Conf. on Management of Data. pp. 322–331 (1990)

5. Bellman, R.: Adaptive Control Processes: A Guided Tour. Princeton University Press (1961)
6. Bentley, J.L.: Multidimensional Binary Search Trees Used for Associative Searching. Communications of the ACM **18**(9), 509–517 (1975)
7. Böhm, C., Braunmüller, B., Krebs, F., Kriegel, H.P.: Epsilon grid order: An algorithm for the similarity join on massive high-dimensional data. In: Proc. of the ACM SIGMOD Intl. Conf. on Management of Data. pp. 379–388 (2001)
8. Böhm, C., Noll, R., Plant, C., Zherdin, A.: Index-supported Similarity Join on Graphics Processors. pp. 57–66 (2009)
9. Comer, D.: The Ubiquitous B-Tree. ACM Comput. Surv. **11**(2), 121–137 (1979)
10. Finkel, R.A., Bentley, J.L.: Quad Trees: A Data Structure for Retrieval on Composite Keys. Acta Informatica **4**(1), 1–9 (1974)
11. Gallet, B., Gowanlock, M.: Load Imbalance Mitigation Optimizations for GPU-Accelerated Similarity Joins. Proc. of the 2019 IEEE Intl. Parallel and Distributed Processing Symp. Workshops pp. 396–405 (2019)
12. Gowanlock, M.: KNN-Joins Using a Hybrid Approach: Exploiting CPU/GPU Workload Characteristics. In: Proc. of the 12th Workshop on General Purpose Processing Using GPUs. pp. 33–42 (2019)
13. Gowanlock, M., Karsin, B.: Accelerating the similarity self-join using the GPU. Journal of Parallel and Distributed Computing **133**, 107 – 123 (2019)
14. Guttman, A.: R-Trees: A Dynamic Index Structure for Spatial Searching. SIGMOD Rec. **14**(2), 47–57 (1984)
15. Kalashnikov, D.V.: Super-EGO: Fast Multi-Dimensional Similarity Join. The VLDB Journal **22**(4), 561–585 (2013)
16. Kim, J., Jeong, W., Nam, B.: Exploiting Massive Parallelism for Indexing Multi-Dimensional Datasets on the GPU. IEEE Transactions on Parallel and Distributed Systems **26**(8), 2258–2271 (2015)
17. Kim, J., Kim, S.G., Nam, B.: Parallel Multi-Dimensional Range Query Processing with R-Trees on GPU. Journal of Parallel and Distributed Computing **73**(8), 1195–1207 (2013)
18. Kim, J., Nam, B.: Co-processing Heterogeneous Parallel Index for Multi-dimensional Datasets. Journal of Parallel and Distributed Computing **113**, 195 – 203 (2018)
19. Lieberman, M.D., Sankaranarayanan, J., Samet, H.: A Fast Similarity Join Algorithm Using Graphics Processing Units. In: 2008 IEEE 24th Intl. Conf. on Data Engineering. pp. 1111–1120 (2008)
20. MIT Haystack Observatory: Space Weather Datasets, ftp://gemini.haystack.mit.edu/pub/informatics/dbscandat.zip, accessed: Feb. 27, 2020
21. Prasad, S.K., McDermott, M., He, X., Puri, S.: GPU-based Parallel R-tree Construction and Querying. In: 2015 IEEE Intl Parallel and Distributed Processing Symp. Workshops. pp. 618–627 (2015)
22. Sellis, T., Roussopoulos, N., Faloutsos, C.: The R+-Tree: A Dynamic Index For Multi-Dimensional Objects. In: Proc. of the 13th VLDB Conf. pp. 507–518 (1987)
23. Shahvarani, A., Jacobsen, H.A.: A Hybrid B+-Tree As Solution for In-Memory Indexing on CPU-GPU Heterogeneous Computing Platforms. In: Proc. of the Intl. Conf. on Management of Data. pp. 1523–1538 (2016)
24. Yan, Z., Lin, Y., Peng, L., Zhang, W.: Harmonia: A High Throughput B+Tree for GPUs. In: Proc. of the 24th Symp. on Principles and Practice of Parallel Programming. pp. 133–144 (2019)