Data Science and Engineering (2021) 6:39–62
https://doi.org/10.1007/s41019-020-00145-x

Heterogeneous CPU-GPU Epsilon Grid Joins: Static and Dynamic Work Partitioning Strategies

Benoit Gallet[1] · **Michael Gowanlock**[1]

Received: 23 June 2020 / Revised: 5 September 2020 / Accepted: 5 October 2020 / Published online: 21 October 2020

Abstract
Given two datasets (or tables) A and B and a search distance ϵ, the distance similarity join, denoted as $A \ltimes_\epsilon B$, finds the pairs of points (p_a, p_b), where $p_a \in A$ and $p_b \in B$, and such that the distance between p_a and p_b is $\leq \epsilon$. If $A = B$, then the similarity join is equivalent to a similarity self-join, denoted as $A \bowtie_\epsilon A$. We propose in this paper Heterogeneous Epsilon Grid Joins (HEGJoin), a heterogeneous CPU-GPU distance similarity join algorithm. Efficiently partitioning the work between the CPU and the GPU is a challenge. Indeed, the work partitioning strategy needs to consider the different characteristics and computational throughput of the processors (CPU and GPU), as well as the data-dependent nature of the similarity join that accounts in the overall execution time (e.g., the number of queries, their distribution, the dimensionality, etc.). In addition to HEGJoin, we design in this paper a dynamic and two static work partitioning strategies. We also propose a performance model for each static partitioning strategy to perform the distribution of the work between the processors. We evaluate the performance of all three partitioning methods by considering the execution time and the load imbalance between the CPU and GPU as performance metrics. HEGJoin achieves a speedup of up to 5.46× (3.97×) over the GPU-only (CPU-only) algorithms on our first test platform and up to 1.97× (12.07×) on our second test platform over the GPU-only (CPU-only) algorithms.

Keywords HEGJoin · Work partitioning · Heterogeneous CPU-GPU computing · Range query · Similarity join · SUPER-EGO

1 Introduction

Consider two input datasets A and B, and a distance threshold ϵ. A distance similarity search finds the pairs of points (p_a, p_b), $p_a \in A$ and $p_b \in B$, such that the distance between these two points is $\leq \epsilon$. While any distance function can be used, in the literature, the Euclidean distance is typically employed [1–6]. These similarity searches are typically computed as a semi-join operation ($A \ltimes_\epsilon B$), where A is a set or table of query points and B a set or table of entries in an index. The particular case where $A = B$ is a self-join (and thus $A \bowtie_\epsilon A$). For simplicity, we examine in this paper the self-join problem. However, we do not explore optimizations exclusive to the self-join. Thus, our optimizations apply to the semi-join case as well. For an input dataset, D, the brute-force self-join solution has a time complexity of $O(|D|^2)$. This complexity decreases when a data indexing method is used to prune the search space. Hence, using an index and the *search-and-refine* strategy, for each query point in D, the *search* of the index generates a set of candidate points that are likely to be within ϵ of the query point, while the *refine* step computes the distance between a query point and its candidate points to produce the final result set.

The indexing methods used for the search-and-refine strategy are often designed for either low [2–4, 6] or high dimensionality [5, 7, 8]. Due to the *curse of dimensionality* [3, 9], when dimensionality increases, index searches become more exhaustive, and the complexity of the algorithm gradually degrades into a brute-force search. Hence, indexes suited for low-dimensional data are likely not to be as efficient when used on higher-dimensional data (and vice versa). The curse of dimensionality is thus among the reasons why we only focus here on the low-dimensionality case, rather than any dimensionality: we elect to create an efficient

✉ Benoit Gallet
benoit.gallet@nau.edu

Michael Gowanlock
michael.gowanlock@nau.edu

[1] School of Informatics, Computing and Cyber Systems, Northern Arizona University, 1295 S Knoles Dr, Flagstaff, AZ 86011, USA

algorithm for the low-dimensional case, rather than a less efficient algorithm that addresses all dimensionalities. Furthermore, while low-dimensional searches are often memory-bound, high-dimensional searches are usually compute-bound, as the cost of a distance calculation increases with dimensionality. In this paper, we focus on low-dimensional searches. Hence, HEGJoin may saturate memory bandwidth, thus potentially negatively impacting performance and parallel scalability of the algorithm, as compared to when fewer processors contend for memory bandwidth.

Graphics processing units (GPUs) have been increasingly used for general computational problems and particularly for improving similarity join performance [4, 5], and with specific data indexing methods that are suited to the GPU's particular single instruction multiple threads (SIMT) architecture [10–14]. The proliferation of GPUs is particularly explained by their increased computational throughput and higher memory bandwidth compared to CPUs. However, despite these attractive features, their use in combination with the CPU to perform some part of the computation to further improve database query throughput, such as the distance similarity join, remains underexplored. Thus, we propose in this paper HEGJoin, a heterogeneous CPU-GPU distance similarity search algorithm. Hence, in addition to the CPU performing GPU-supporting tasks (launching kernels, transferring data, etc.), we explicitly use the CPU to compute a fraction of the total number of query points.

As discussed above, the literature concerning heterogeneous CPU-GPU database applications is relatively scarce. Thus, we propose to leverage both the CPU and GPU and design an efficient algorithm to compute distance similarity searches. There are two major CPU-GPU similarity search algorithm designs, described as follows:

- *Task parallelism* Assign the CPU and GPU particular tasks to compute, such as *searching* on the CPU and then *refining* on the GPU [15].
- *Data parallelism* Split the data to compute and perform both the *search* and *refine* steps on each architecture independently, using different algorithms suited to the strengths of each architecture [16].

In the literature, heterogeneous CPU-GPU similarity search and related range query algorithms focus on a *task-parallel* approach [15, 17]. The *task-parallel* approaches model the problem as a two [15] or three stage pipeline [17], where the CPU is assigned one task, such as searching an index, and the GPU is assigned the task of refining the candidate points [15]. Consequently, as with any pipeline, the throughput is dependent on the slowest stage. Therefore, the drawback of the *task-parallel* approach is that it can leave resources (CPU or GPU) underutilized. In this paper, we focus on the data-parallel approach, which allows us to exploit all available computational resources in the system to maximize query throughput. Since we concurrently use the CPU and GPU, we then need to efficiently partition the work among our processors, i.e., assign to each processor a number of queries to compute so the algorithm achieves good load balancing and thus good performance. To the best of our knowledge, HEGJoin is the first *data-parallel* heterogeneous and concurrent CPU-GPU distance similarity join algorithm.

As our solution is designed for data-parallelism, our work partitioning strategies partition queries from the input dataset. Because HEGJoin is a heterogeneous CPU-GPU algorithm, this is particularly challenging as we need to efficiently distribute the work to accommodate each processor's architectural characteristics. The data-parallel work partitioning can be achieved by different methods: dynamically [16] where the work is assigned to the processors on-demand, or statically [18, 19], where each processor has a fixed amount of work to compute. Statically partitioning the work is challenging, as we need to determine the amount of work to be assigned to the CPU and GPU such that it minimizes load imbalance between the processors. The workload has data-dependent performance characteristics that depend on the number of points, their dimensionality, and their distribution (e.g., underdense vs. overdense regions). Consider partitioning the data using a dynamic approach. In this case, partitioning involves having the pieces of work assigned to the CPU or the GPU, where there is a trade-off between small work units assigned to each processor to achieve good load balancing, and large work units so that the processors reach peak throughput. On the other hand, static partitioning requires accurately estimating the total workload, which is particularly challenging given the data-dependent nature of the work. In contrast, on other problems that have deterministic workloads, the workload can be accurately estimated, and static work partitioning is straightforward [20].

To enable static partitioning, we propose two performance models that quantify the workload based on different metrics that enable the two static partitioning strategies to assign work to the CPU and GPU. Additionally, we propose a dynamic partitioning strategy that is oblivious to the workload. We compare these partitioning strategies to assess their relative strengths and weaknesses, to understand how the characteristics of the workload may affect the performance of HEGJoin, and to ultimately be able to select the partitioning strategy that yields the best performance.

Our algorithm leverages two previously proposed independent works that were shown to be highly efficient: the GPU algorithm (LBJoin) by [6] and the CPU algorithm (Super-EGO) by [3]. However, although we mention above that HEGJoin employs a data-parallel approach, as we leverage two different algorithms (LBJoin and Super-EGO) and a work queue, our algorithm also has task-parallel

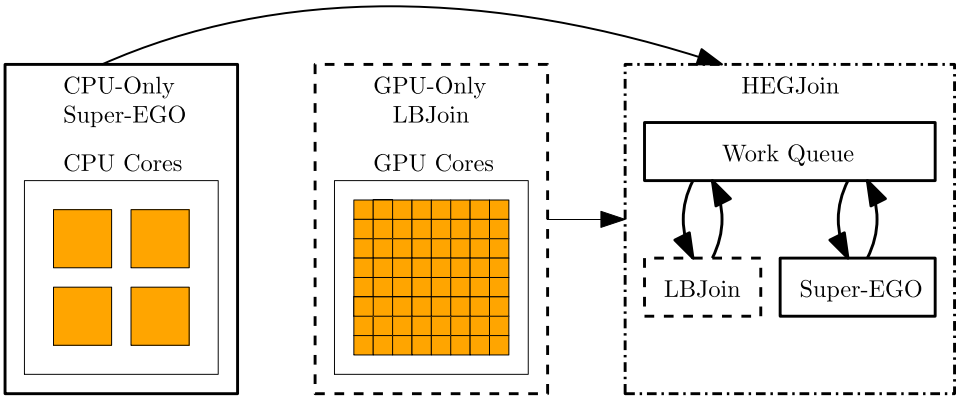


Fig. 1 Representation of how we combine SUPER-EGO and LBJOIN by using a single work queue to form HEGJOIN. When using the static partitioning strategy, the CPU and the GPU would access the work queue only once (at the beginning of the algorithm to retrieve their assigned queries). When using the dynamic partitioning scheme, the CPU and the GPU would iteratively query the work queue for queries to compute until it is empty

characteristics. While the output of LBJOIN and SUPER-EGO is identical, the algorithm executed by the CPU is inherently different from the algorithm executed by the GPU. Therefore, HEGJOIN uses a mixed parallelism model (a combination of data- and task-parallelism). Figure 1 illustrates how LBJOIN and SUPER-EGO work together through the use of a single shared work queue.

By combining the LBJOIN and SUPER-EGO algorithms and using our work partitioning methods, we achieve better performance on most experimental scenarios than CPU-only or GPU-only approaches. Note that, since SUPER-EGO and LBJOIN respective indexing methods are more efficient in lower dimensions, and as most of the related literature works rarely focus on both low and high dimensionality, we choose to focus on low-dimensional distance similarity joins. Hence, this paper makes the following contributions:

1. We combine state-of-the-art algorithms for the CPU and GPU to propose a new algorithm, HEGJOIN, and which is, to the best of our knowledge, the first data-parallel heterogeneous and concurrent CPU-GPU distance similarity join algorithm.
2. We propose an efficient shared double-ended work queue (deque) to assign query points either to the CPU or to the GPU. Furthermore, we exploit the GPU's high computational throughput by assigning it query points with the highest workload (located at the beginning of the deque), while we assign the query points with the smallest workload to the CPU.
3. We develop three different workload partitioning strategies. The dynamic work partitioning strategy uses the shared deque to assign work to either the CPU or GPU. In this case, there is no fixed boundary on the work that can be assigned to the CPU or the GPU, as it is assigned to processors on-demand. Furthermore, we advance two static work partitioning methods: based on the number of query points and based on the total number of candidate points that need to be refined per query point. As with both static strategies, the CPU and GPU have a fixed number of queries to compute, if the GPU completes its work before the CPU, it must wait for the CPU to complete its work (and vice versa).
4. We optimize SUPER-EGO to further improve the performance of HEGJOIN. We denote this optimized version of SUPER-EGO as NEW-SUPER-EGO.
5. We evaluate the performance of HEGJOIN using seven real-world and ten exponentially distributed synthetic datasets and using two platforms. We achieve speedups up to 5.46× and 3.97× over the GPU-only and CPU-only algorithms on the first test platform and speedups up to 1.97× and 12.07× on the second test platform. Furthermore, we achieve an average load imbalance ratio as low as 0.14 when using the dynamic work partitioning strategy on the first platform.

The paper is organized as follows. We begin in Sect. 2 by surveying the literature and presenting an overview of GPU architecture. We then present in Sect. 3 the leveraged algorithms, and we describe HEGJOIN and its main features in Sect. 4. We evaluate the performance of HEGJOIN and our partitioning methods in Sect. 5, and we finally conclude this paper in Sect. 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 $\text{dist}(q, c) \leq \epsilon$, where $\text{dist}(q, c) = \sqrt{\sum_{j=1}^{d}(q(j) - c(j))^2}$. 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 GPU Architecture

We present material related to GPU architecture and use CUDA terminology throughout the paper. Modern GPUs are equipped with a few thousand cores. The global memory bandwidth of the GPU is over an order of magnitude higher than the main memory bandwidth of the CPU (up to 1555 GB/s for the Tesla A100 [21] GPU). However, the GPU's global memory has limited capacity, and the potential for parallelism is dependent on control flow, as threads are executed in groups of 32 (called *warps*) in lock step. Also, different workloads assigned to threads within the same warp induce idle periods, where some threads are idle while others are computing. The PCI interconnect between the CPU and the GPU is a bottleneck (PCIe-v3 has 32 GiB/s bidirectional bandwidth). For more information on the CUDA programming model and the GPU architecture, we refer the reader to general references on the topic [22, 23].

2.3 Related Work

In this section, we outline relevant work regarding the distance similarity join and work partitioning methods between heterogeneous architectures.

2.3.1 Data Indexing

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 the existing literature aims at improving performance using either the CPU or the GPU, and rarely both. Hence, the literature regarding heterogeneous CPU-GPU similarity join optimizations remains relatively scarce. The search-and-refine strategy (Sect. 1) largely relies on the use of data indexing methods that we describe as follows.

Indexing data structures are used to prune the search space of an indexed input dataset to reduce the number of candidates that may be within ϵ of each query point. 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 both architectures. We identify different indexing methods, including those designed for the CPU [2, 3, 24–29], the GPU [10, 12, 30], or both architectures [15–17]. As our algorithm focuses on the low-dimensionality distance similarity search, we focus on presenting indexing methods that are 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 [3, 6] that both use a grid index 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 used to index multi-dimensional data are based on trees. The following trees have been designed for range queries and can, therefore, be used for distance similarity searches. The *k*D-Tree [24] is a binary tree that indexes *k*-dimensional data by subsequently splitting the search space in two, following an alternation of the *k* dimensions (in two dimensions for example, split following the *x*-axis, then the *y*-axis, then the *x*-axis, etc.). Hence, each node stores the coordinates of its search space and splits it between its two child nodes. The Quad Tree [31] is very similar to the *k*D-Tree, as it consists of a tree whose nodes have four children, and as the search space is subsequently divided into four subspaces (instead of two for the *k*D-Tree). The nodes of the R-Tree [27] consist of bounding boxes to store multi-dimensional objects, which are then stored in the leaf nodes of the tree. In addition to these tree indexes, grids such as the Epsilon Grid Order (EGO) [2, 3] have also been designed for distance similarity joins. We discuss this EGO index that we leverage in Sect. 3.2.

GPU Indexing Similar to CPU indexes, index-trees have been optimized to address the GPU's SIMT architecture. [12] optimized the R-Tree on the GPU by replacing the recursive accesses inherent to traversing the tree that are not suited to the GPU. They replaced these accesses by sequential accesses, particularly by allowing the search of the tree to jump from a node to its next sibling. [10] improve the efficiency of the B-Tree by using nodes the size of the GPU's cache access size and by avoiding recursive calls during the tree traversal as well. Furthermore, they assign multiple queries to a warp, with all the threads of the same warp that cooperate to compute one query at a time, thus reducing intra-warp thread divergence. We leverage the GPU

grid index proposed by [30] and that is designed for distance similarity joins, which we present in Sect. 3.1.1.

CPU-GPU Indexing [15] propose an R-Tree designed for range queries that uses task parallelism. The CPU searches the internal nodes of the tree and, when reaching the leaf nodes, sends this partial result to the GPU. The GPU then traverses these leaf nodes, which are stored as a contiguous array in GPU's main memory so the memory accesses are likely coalesced, and refines the candidate objects. In contrast, [16] elects to use two indexes for data parallelism to compute kNN searches. The CPU uses a kD-Tree [24], while the GPU uses a grid [30]. Hence, both indexes are suited to their respective architecture.

2.3.2 Workload Partitioning

As described above, efficiently partitioning the work of parallel algorithms is critical, whether it is based on the tasks to execute (task-parallelism), or the data to compute (data-parallelism). Because the solution we propose in this paper requires data-parallelism, we describe in this section contributions in the literature that propose partitioning schemes for data-parallel algorithms as well.

Efficient work partitioning is usually difficult to achieve since several parameters need to be considered: typically the processors' relative performance (e.g., computational throughput or memory bandwidth), and if the algorithm's workload can be easily determined (usually the case for non data-dependent workloads). On problems exposing a data-dependent workload, such as the distance similarity join or sparse matrix multiplications [32] for example, determining the workload is more challenging than for problems without data-dependent workloads (e.g., regular matrix–matrix multiplications [20]).

Dynamic partitioning solutions [16] present advantages to keeping the processors busy (as they are assigned work until none is available) and do not require knowing the relative performance of the processors beforehand, making it agnostic to platform hardware characteristics. Furthermore, while dynamic work partitioning does not require knowledge about the workload to be functional, it may still be beneficial to determine an overall workload in order to assign work to the most suitable processor.

On the other hand, static partitioning methods [18, 19], if not arbitrary (i.e., a static partitioning of work not based on information related to the processors or the algorithm), requires having accurate knowledge about the relative performance of the processors as well as the workload to achieve good load balancing between the processors. Furthermore, most static partitioning methods are based on models [18, 19], which are made for a specific algorithm and platform. Hence, their solution may be inefficient when used for a different algorithm (which would require a new model) or on a different platform (which would require adapting the model for this new platform).

Michael [16] proposes a dynamic partitioning scheme to compute kNN searches. Using a work queue, they continuously assign query points to the CPU and the GPU until all the work has been computed. As the overall workload of the algorithm is determined beforehand, they are able to assign more query points and with the highest workloads to the GPU and the rest to the CPU. The load balancing of the computation is thus managed by the work queue and the dynamic work assignment to the different processors.

Dominik and O'Boyle [18] advance a general static partitioning scheme of applications. Their solution relies on different metrics such as the number of computing operations and their precision, the number of memory operations, the presence of loops, etc., extracted from the code before the computation. Hence, they determine an overall workload for the algorithm and, if the computation is considered to be efficient if executed on both the CPU and GPU, then they estimate a work partitioning using the input data size, and a model they previously developed using the same application and for several fixed static partitioning fractions. [19] propose a model to statically assign the work of a fast Fourier transform (FFT) to the CPU and the GPU. Their solution creates subproblems of the FFT and, following their model, assign these subproblems to the most suitable processor. This model is based on parameters such as previously recorded performance, CPU-GPU data transfer rate, memory management on the GPU, matrix transposition performance, and several other factors.

The dynamic work partitioning strategy we propose in this paper, while similar to the one proposed by [16], should be more efficient as the way we determine our workload is more accurate than their solution. Our static work partitioning methods, similarly to other static partitionings [18, 19], also propose a performance model (for each of our static partitioning strategy). However, we outline the importance of determining the overall workload to efficiently partition, by proposing an intuitive solution having rather little knowledge about the workload, and a second method with an accurate knowledge of the workload. The load imbalance between the CPU and GPU would show such importance. For comparative purposes, we expect that our solution with accurate workload knowledge will yield better load balancing than the solution with less knowledge of the workload.

3 Leveraged Work

In this section, we present the leveraged works used to design HEGJoin. We use LBJoin [6] for the GPU and Super-EGO [3] for the CPU, which are two state-of-the-art algorithms for their respective platforms, which are publicly

available. For greater detail, we encourage the reader to refer to the original papers of SUPER-EGO [3] and LBJOIN [6]. Furthermore, we acknowledge that a CPU distance similarity join algorithm has been proposed in the literature by [33] that outperforms SUPER-EGO at high dimensions. However, their algorithm has comparable performance to SUPER-EGO in low dimensionality. Therefore, we use SUPER-EGO and not [33] to create HEGJOIN, as it is better suited to our low-dimensional case.

3.1 GPU Algorithm: LBJOIN

The GPU component of HEGJOIN is based on the GPU kernel proposed by [6]. This kernel also uses the grid index and the batching scheme by [30]. 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 Sect. 2.)

3.1.1 Grid Indexing

The grid index presented by [30] 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 in the GPU's global memory: (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, and (iii) the third array corresponds to the position of the points in the dataset and is pointed to by the second array. Candidate points are retrieved by searching the index in global memory, which yields a set of candidates points in the dataset, D. Furthermore, the threads within the same warp access adjacent 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.

3.1.2 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 [30], 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 $n_b = n_e/n_s$.[1] The buffer size, n_s, can be selected such that the GPU's global memory capacity is not exceeded. The number of query points, n_p^{GPU}, processed per batch (a fraction of $|D|$) are defined by the number of batches as follows: $n_p^{\text{GPU}} = |D|/n_b$. Hence, a smaller number of batches will yield a larger number of queries processed per batch.

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.

3.1.3 Sort by Workload and Work Queue

The sorting strategy proposed by [6] 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. Hence, the grid index is used to retrieve the adjacent cells and to find the number of points in each of them. 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 [6], 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. Furthermore, the sum of the individual workloads of each query point corresponds to the total workload. Since this sorting by workload strategy uses the grid index to compute the workload, it allows for estimating the workload for any input dimensionality and data distribution.

3.1.4 GPU Kernel

The GPU kernel [6] makes use of a grid index, the batching scheme, as well as the sorting by workload strategy and the

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

work queue presented above. Moreover, we configure the kernel [6] 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 candidate 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 HEGJoin is based on the efficient distance similarity join algorithm, Super-EGO, proposed by [3]. We detail its main features as follows.

3.2.1 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.

3.2.2 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 [2].

3.2.3 Join Method

The Super-EGO algorithm takes a set of query points and computes each point's result set as follows. First, in main memory, Super-EGO recursively creates new partitions, until these partitions reach a given size. Next, the join is made by comparing the set of query points to this set of generated partitions of the input dataset, where the partitions that are recursively generated are sets of points spatially co-located to the set of query points. 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.

3.2.4 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

In this section, we present the major components of our heterogeneous CPU-GPU algorithm, HEGJoin, the different techniques we propose to partition the workload between the CPU and the GPU, as well as improvements made to the work we leverage.

4.1 Shared Work Queue

As mentioned in Sect. 3.1, we leverage the work queue stored on the GPU that was proposed by [6], which efficiently balances the workload between GPU threads. However, to use the work queue for the CPU and the GPU components of HEGJoin, 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 [16] for the CPU-GPU kNN algorithm, the query points need to be sorted based on their workload, as detailed in Sect. 3.1.3. However, while query points' workload in [16] 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 [16], 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 [6] 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 the CPU's workload increases, as their respective index progresses in the

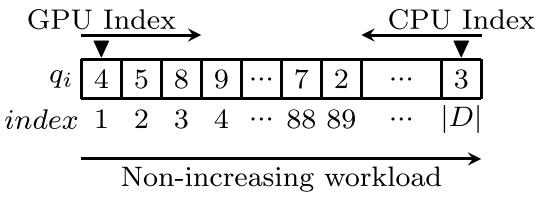


Fig. 2 Representation of our deque as an array. The numbers q_i are the query points id, the triangles are the starting position of each index, and the arrows above it indicate the indices progression in the deque

deque. Also, note that while n_p for the CPU (n_p^{CPU}) is fixed, n_p for the GPU (n_p^{GPU}) varies based on the dataset characteristics and on ϵ (Sect. 3.1.2).

As described in Sect. 3, HEGJoin uses two different sorts: sorting by workload (Sect. 3.1.3) and Super-EGO 's EGO-sort (Sect. 3.2.2). 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 Fig. 3.

4.2 Workload Partitioning

As previously described, this paper proposes three different methods to partition the work between the CPU and the GPU, using the shared work queue presented in Sect. 4.1. Proposing these three methods allows us to extensively explore work partitioning characteristics, as well as demonstrate the significance of an efficient work partitioning method. Our three partitioning methods use the shared deque presented in Sect. 4.1, as in all three cases the work still needs to be partitioned among threads. Thus, we describe these partitioning strategies as follows.

4.2.1 Dynamic Work Partitioning Strategy

Our dynamic work partitioning strategy assigns work to the CPU and GPU on-demand until the queue is empty. Constantly querying the queue for a fraction of work provides good load balancing, as the processors are likely to complete their last batch of queries at roughly the same time. The CPU and GPU are both assigned a batch size large enough to accommodate their relative performance (particularly for the GPU, to achieve good occupancy), as well as to reduce the number of atomic accesses to the queue. However, the batch size for the CPu and GPu is also not too large, so they are not assigned too many query points as it might leave a processor without work to compute while the other one is computing a large batch. Figure 2 illustrates how the dynamic partitioning strategy works. We describe the procedure used to assign the query points to the processors as follows:

1. We set the GPU's deque index to 1 and the CPU's deque index to |*D*|.
2. The program terminates if the GPU's and CPU's indices are at the same position in the deque.
3. To assign query points to a GPU stream, we create and assign a new batch of queries to this GPU stream and increase GPU's deque index.
4. To assign query points to CPU thread, we create and assign a new batch of queries to this CPU thread and decrease CPU's deque index.

4.2.2 Static Partitioning Strategy Based on Query Points

This static work partitioning method splits the number of query points between the processors, using a static partitioning fraction p_q, where $0 \leq p_q \leq 1$. Hence, from p_q and a number of query points (|*D*|), we can determine the number of query points n_q^{GPU} to assign to the GPU and, by extension, to the CPU. This partitioning fraction p_q is determined based on the estimation of the workload w_{est}, as a function of the number of query points and ϵ. We consider for this partitioning strategy the equal workload assumption that we describe as follows.

Equal Workload Assumption In this model, we assume that we do not know the workload of each query point. Thus, we consider that each query point has the same

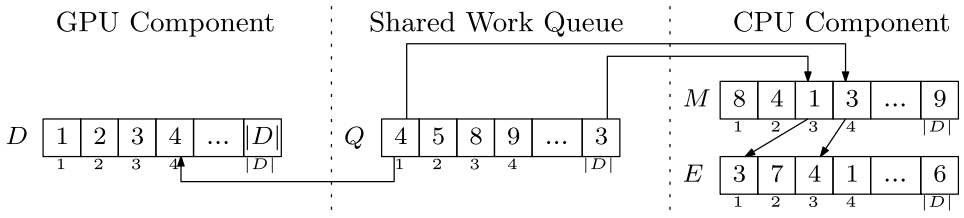


Fig. 3 Illustration of an input dataset *D*, the shared deque sorted by workload *Q*, the input dataset EGO-sorted *E* and the mapping *M* between *Q* and *E*. The numbers in *D*, *Q*, and *E* correspond to query point ids, while the numbers in *M* correspond to their position in *E*. The numbers below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

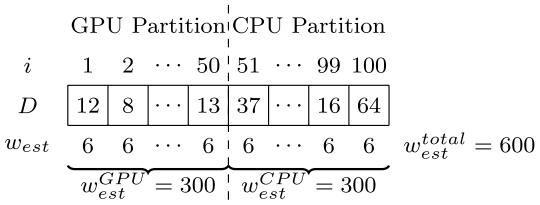


Fig. 4 Representation of the static partitioning strategy based on the query points, where D is the dataset sorted by workload, i refers to the indices of the query points in D, and W_{est} the workload of the points using the equal workload assumption and that is determined by our model (and where $w_{\text{est}}^{\text{total}}$ is the estimated workload of HEGJoin). As we use the equal workload assumption, each query point is therefore assigned the same workload. Furthermore, we consider in this example that the model considers the CPU and GPU to have the same throughput, and thus assigns the same estimated workload to both processors ($w_{\text{est}}^{\text{CPU}} = w_{\text{est}}^{\text{GPU}}$), i.e., the same number of query points

workload. For example, if data are largely unstructured, similarly to a uniform distribution, then all query points would have roughly the same amount of work to compute. Then, based on the query throughput of the CPU and GPU, we assign each architecture a fraction of the total number of queries.

Figure 4 illustrates the static partitioning strategy based on query points. Using the equal workload assumption, this example shows the case where the model assigns the same number of query points to the CPU and GPU. In this example, we have an input dataset D sorted by workload (Sect. 3.1.3) made of 100 query points ($|D| = 100$) and indexed by i. In this example, our model determines an overall workload $w_{\text{est}}^{\text{total}} = 600$, which, following the equal workload assumption, corresponds to each point having an estimated workload $w_{\text{est}} = w_{\text{est}}^{\text{total}}/|D| = 6$. We consider in this example that the model determines that the CPU and GPU have the same throughput and should therefore be assigned the same workload ($w_{\text{est}}^{\text{CPU}} = w_{\text{est}}^{\text{GPU}}$), and thus the same number of query points. Depending on the dataset's characteristics (such as its distribution), this estimated workload might differ from the actual workload to compute, which may have an impact on the overall performance of HEGJoin when using such a static partitioning strategy, compared to the other partitioning strategies we propose in this paper.

Using the equal workload assumption, we propose a model to determine the static partitioning fraction p_q for HEGJoin (where $0 \leq p_q \leq 1$). For a specific dataset in d dimensions, we consider a reference search distance ϵ_r, its search volume in d dimensions $v(\epsilon_r) = \frac{\pi^{d/2}}{\Gamma(\frac{d}{2}+1)} \times {\epsilon_r}^d$, and the corresponding execution time of LBJoin ($T_{\epsilon_r}^{\text{GPU}}$) and Super-EGO ($T_{\epsilon_r}^{\text{CPU}}$) when computing the distance similarity join on d with the reference search distance ϵ_r. Hence, for a given search distance ϵ_s for which we want to determine the work partitioning fraction p_q, we predict the execution time of LBJoin and Super-EGO by scaling their execution time $T_{\epsilon_r}^{\text{GPU}}$ and $T_{\epsilon_r}^{\text{CPU}}$ by the ratio of the search volume $v(\epsilon_s)$ over the reference search volume $v(\epsilon_r)$. The ratio $v(\epsilon_s)/v(\epsilon_r)$ corresponds to the estimated workload increase when the search distance increases as well. Thus, we predict the execution time of the GPU-only algorithm (LBJoin) T^{GPU} as follows:

$$T^{\text{GPU}}(\epsilon_s, \epsilon_r, T_{\epsilon_r}^{\text{GPU}}) = T_{\epsilon_r}^{\text{GPU}} \times \frac{v(\epsilon_s)}{v(\epsilon_r)} \tag{1}$$

Similarly, we predict the execution time of the CPU-only algorithm (Super-EGO) T^{CPU} as follows:

$$T^{\text{CPU}}(\epsilon_s, \epsilon_r, T_{\epsilon_r}^{\text{CPU}}) = T_{\epsilon_r}^{\text{CPU}} \times \frac{v(\epsilon_s)}{v(\epsilon_r)} \tag{2}$$

We then compute the GPU query throughput (the number of query points the GPU can process per second) as $f_q^{\text{GPU}} = |D|/T^{\text{GPU}}(\epsilon_s, \epsilon_r, T_{\epsilon_r}^{\text{GPU}})$, as well as the CPU query throughput $f_q^{\text{CPU}} = |D|/T^{\text{CPU}}(\epsilon_s, \epsilon_r, T_{\epsilon_r}^{\text{CPU}})$. In addition, we consider the upper bound query throughput as $f_q = f_q^{\text{GPU}} + f_q^{\text{CPU}}$, and which corresponds to the sum of the GPU and CPU query throughput. Using this upper bound query throughput f_q, we can predict the execution time T^{HEGJoin} of HEGJoin when using any of our static partitioning strategies. We compute this predicted execution time as follows:

$$T^{\text{HEGJoin}} = |D|/f_q \tag{3}$$

In addition to predicting the execution time of HEGJoin, we use the upper bound query throughput f_q to determine the static partitioning fraction p_q as the ratio of f_q^{GPU} over f_q. Consequently, we compute the static partitioning fraction as follows:

$$p_q = f_q^{\text{GPU}}/f_q \tag{4}$$

Finally, we use p_q to determine the number of query points to assign to the GPU as $n_q^{\text{GPU}} = |D| \times p_q$. By extension, we determine the number of query points to assign to the CPU as $n_q^{\text{CPU}} = |D| - n_q^{\text{GPU}}$.

4.2.3 Static Partitioning Strategy Based on Candidate Points

Static partitioning based on candidate points considers the total number of candidate points to refine, as well as the number of candidate points of the individual query points. Hence, while the previous static partitioning strategy assumes an equal workload between the query points, we acknowledge here that the query points are likely to each

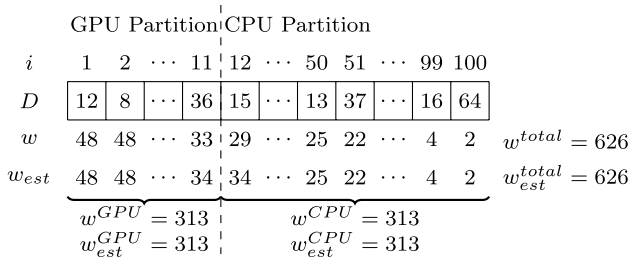


Fig. 5 Representation of the static partitioning strategy based on the candidate points, where D is the dataset, i the indices of the query points in D, the workload of the points w as used to sort them by their workload (and where w^{Total} is the total number of candidate points to refine), and w_{est} the workload of the query points determined by the model (and where $w_{\text{est}}^{\text{total}}$ is the total estimated workload of HEGJoin). While we consider for this example that the model estimates a workload that is equal to the workload of HEGJoin ($w_{\text{est}}^{\text{total}} = w^{\text{Total}}$), there might be scenarios in which $w_{\text{est}}^{\text{total}}$ and w^{Total} are different. We consider in this example that the model estimates the CPU and GPU to have the same throughput and thus assign the same number of estimated candidate points to refine to the CPU and to the GPU ($w_{\text{est}}^{\text{CPU}} = w_{\text{est}}^{\text{GPU}}$). Furthermore, as in this example the estimated workload is the same as the actual workload of HEGJoin ($w_{\text{est}}^{\text{total}} = w^{\text{Total}}$), both processors are assigned the same amount of work to compute ($w^{\text{GPU}} = w^{\text{CPU}}$), i.e., the same number of candidate points to refine

have a different workload. We thus propose the unequal workload assumption as follows.

Unequal Workload Assumption We consider for this model that each query point can have a workload different from the other query points. Hence, if a dataset has dense regions and sparse regions, the workload that is assigned to the query points is an accurate reflection of their workload in comparison to the *equal workload assumption*.

Figure 5 illustrates the static partitioning strategy based on the number of candidate points to refine. In this example, the model considers that the CPU and GPU have the same throughput and should, therefore, be assigned the same number of candidate points to refine. Hence, we have an input dataset D with 100 query points ($|D| = 100$) that are sorted by their respective workload w, and a total number of candidate points to refine $w^{\text{total}} = 626$. This model estimates a number of candidate points to refine $w_{\text{est}}^{\text{total}} = 626$, which is split equally between the CPU and the GPU (as the model considers they have the same throughput in this example). Thus, the GPU is assigned an estimated total number of candidate points to refine $w_{\text{est}}^{\text{GPU}} = 313$, and $w_{\text{est}}^{\text{CPU}} = 313$ for the CPU. And, since this model considers the unequal workload assumption, the GPU's workload is the same as its estimated workload ($w^{\text{GPU}} = w_{\text{est}}^{\text{GPU}}$). The same outcome applies to the CPU ($w^{\text{CPU}} = w_{\text{est}}^{\text{CPU}}$). Furthermore, while this example workload corresponds to 11 query points for the GPU and 89 query points for the CPU (determined by the cumulative workload of these query points), the respective total number of candidate points to refine of the GPU and the CPU is similar ($w^{\text{GPU}} \approx w^{\text{CPU}}$). Given that the CPU and GPU are considered to have the same throughput in this example, this strategy should yield a relatively low load imbalance between the CPU and GPU.

This static partitioning strategy uses Equations 1 and 2 to predict the execution time of LBJoin and Super-EGO for a specific dataset and a given search distance ϵ_s. Hence, we use the total number of candidate points to refine w, as determined when sorting the query points by their workload, in addition to the predicted execution time $T^{\text{GPU}}(\epsilon_s, \epsilon_r, T_{\epsilon_r}^{\text{GPU}})$ to compute the GPU candidate point throughput (the number of candidate points the GPU can refine per second) $f_c^{\text{GPU}} = w/T^{\text{GPU}}(\epsilon_s, \epsilon_r, T_{\epsilon_r}^{\text{GPU}})$. Similarly to the GPU, we compute the number of candidate points throughput refined by the CPU $f_c^{\text{CPU}} = w/T^{\text{CPU}}(\epsilon_s, \epsilon_r, T_{\epsilon_r}^{\text{CPU}})$. In comparison with the static partitioning based on the query points (Sect. 4.2.2), we compute here the upper bound throughput of the number of candidate points refined $f_c = f_c^{\text{GPU}} + f_c^{\text{CPU}}$, which corresponds to the sum of the throughput of the number of candidate points refined by the CPU and GPU. We then use this upper bound candidate refinement throughput to determine the static partitioning fraction p_c (where $0 \leq p_c \leq 1$), and which is computed as follows:

$$p_c = f_c^{\text{GPU}}/f_c \tag{5}$$

We then use this static partitioning fraction p_c to determine the number of candidate points to assign to the GPU, $n_c^{\text{GPU}} = w \times p_c$. Similarly, we determine the number of candidate points to assign the CPU, $n_c^{\text{CPU}} = w - n_c^{\text{GPU}}$. Furthermore, as we consider the unequal workload assumption we described above, we need to find the number of query points to assign to the GPU, n_q^{GPU}, and for which their cumulative workload is the closest to the GPU's assigned workload n_c^{GPU} (by extension, we also find $n_q^{\text{CPU}} = |D| - n_q^{\text{GPU}}$).

While the GPU and CPU do not use the same indexing method, and thus do not yield the same number of candidate points to refine, our experimental evaluation (Sect. 5) will show that the number of candidate points to refine, w, yielded by the grid indexing schemes in LBJoin (Sect. 3.1) and Super-EGO (Sect. 3.2) are roughly consistent such that we assume w is equal for both indexing schemes.

4.2.4 Summary of Work Partitioning Strategies

In this section, we summarize the key points of the work partitioning strategies we presented in Sects. 4.2.1, 4.2.2 and 4.2.3 above.

- *Dynamic Partitioning Strategy* This work partitioning strategy uses the shared deque proposed in Sect. 4.1 to

assign query points to the CPU and GPU on-demand, until the deque is empty. The main objective of this partitioning method is to have the CPU and GPU finishing their last batch of query points roughly at the same time, particularly by frequently querying the deque for a new batch to compute. We denote HEGJoin using this dynamic partitioning strategy as HEGJoin-Dyn.

- *Static Partitioning Strategy Based on Query Points* The static partitioning strategy based on query points that we described in Sect. 4.2.2 estimates the workload of HEGJoin to assign a number of query points to the CPU and GPU. Given a specific dataset, a search distance ϵ_r and the execution time of LBJoin and Super-EGO, this strategy estimates the computation time of HEGJoin by scaling the execution time of LBJoin and Super-EGO using ϵ_r and the search distance used to compute the distance similarity join. From this estimated computation time and the execution time of the GPU-only and CPU-only algorithms, we determine the static partitioning fraction p_q (where $0 \leq p_q \leq 1$), and then the number of query points to assign to the GPU and CPU, assuming that all the query points have an equal workload. We denote HEGJoin using this static partitioning strategy based on query points as HEGJoin-SQ.
- *Static Partitioning Strategy Based on Candidate Points* This static partitioning strategy based on candidate points that we introduced in Sect. 4.2.3 divides the total number of candidate points to refine between the CPU and GPU. Similarly to the partitioning method based on query points, we predict the execution time of LBJoin and Super-EGO the same way as we do for the static partitioning strategy based on query points. However, we use this predicted execution time to determine the static partitioning fraction p_c and splits the total number of candidate points to assign to the CPU and GPU. Hence, we determine the number of candidate points to assign to the GPU and CPU and then find the number of query points whose cumulative workload is the closest to the workload assigned to the GPU. Similarly, we determine the number of query points to assign to the CPU. We denote HEGJoin using this static partitioning strategy based on candidate points as HEGJoin-SC.

Table 1 summarizes the properties of HEGJoin-Dyn, HEGJoin-SQ, and HEGJoin-SC. HEGJoin-Dyn and HEGJoin-SC have mutually exclusive properties, whereas HEGJoin-SQ overlaps the properties of HEGJoin-Dyn and HEGJoin-SC. By examining these three work partitioning strategies, we cover a large range of properties, thus enabling us to make a comprehensive examination of work distribution in HEGJoin.

Table 1 Summary of the different properties of HEGJoin-Dyn, HEGJoin-SQ, and HEGJoin-SC

	HEGJoin-Dyn	HEGJoin-SQ	HEGJoin-SC
Workload-oblivious	✓	✓	
Workload-aware			✓
On-demand	✓		
Planned		✓	✓
Architecture-oblivious	✓		
Architecture-aware		✓	✓

30	30	30	30	**20**	22	22	22	**22**	22	22	22
1	2	3	4	5	6	7	8	9	10	11	12

Fig. 6 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

4.3 Batching Scheme: Complying with Non-Increasing Workload

Because the batching scheme proposed by [30] and presented in Sect. 3.1.2 was not designed for non-increasing workloads, we had to adapt it to fit our sort by workload strategy (Sect. 3.1.3) and its non-increasing workload. Indeed, as the batch estimator creates batches with a fixed number of query points, and because the query points are sorted by workload, this batching scheme creates successive batches with a non-increasing workload. Hence, as the execution proceeds, the batches become smaller, take less time to compute, and the overhead of launching many kernels may become substantial, particularly when the computation could have been executed with fewer batches.

We modify the batching scheme (Sect. 3.1.2) to accommodate the sort by workload strategy, and that is represented in Fig. 6. 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 and avoid buffer overflow during computation). Adding the estimated and the inferred number 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.

When using the dynamic partitioning (Sect. 4.2.1), we set a minimum number of batches to $2 \times n_f$, where $n_f = 3$ 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. When using a static partitioning strategy (Sects. 4.2.2 and 4.2.3), then we set the minimum number of batches for the GPU $n_f = 3$, so each CUDA stream has at least a batch to compute. We do not create more batches for the CPU, as it already has its own reserved fraction of the work, determined by one of the static partitioning strategies.

4.4 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 [6] and presented in Sect. 3.

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

In the kernel, since the work queue has been relocated 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 that is 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.

4.5 CPU Component: HEGJoin-CPU

The CPU component of HEGJoin, which we denote as HEGJoin-CPU, is based on the Super-EGO algorithm proposed by [3] and presented in Sect. 3.2. We make several modifications to Super-EGO to incorporate the shared deque we use, and we also optimize Super-EGO to improve its performance. We denote this improved version of Super-EGO as New-Super-EGO.

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

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 Sect. 4.1 and illustrated in Fig. 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 New-Super-EGO 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, New-Super-EGO is now capable to compute and to store data using 64-bit floats instead of only 32-bit floats.

5 Experimental Evaluation

In this section, we present the experimental evaluation we conducted to measure the performance of HEGJoin against the work it leverages (LBJoin and Super-EGO), as well as the efficiency of the different work partitioning strategies we propose in this paper.

5.1 Selectivity

We report the selectivity as defined by [3] 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. The selectivity thus corresponds to the average number of neighbors found per query point, excluding the query points from finding themselves.

5.2 Datasets

In this section, we present the real-world and synthetic datasets we use to evaluate the performance of HEGJoin and our partitioning strategies. We detail the real-world datasets that we select as follows:

- *SW-* [34], composed of 1.86M or 5.16M points in two dimensions representing the latitude and longitude of the objects, and adding the total number of electrons as the third dimension.
- *SDSS* [35], composed of a sample of 15.23M galaxies in two dimensions.
- *Gaia* [36], in which we select the position of 50M objects from the Gaia catalog.

Table 2 Summary of the datasets used to conduct our experiments. $|D|$ denotes the number of points, d the dimensionality, and S the selectivity range for the values of ϵ we use

Dataset	$\|D\|$	d	S	Dataset	$\|D\|$	d	S
Expo2D2M	2 M	2	397–9.39 K	*Expo2D10M*	10 M	2	80–1.99 K
Expo3D2M	2 M	3	64–6.70 K	*Expo3D10M*	10 M	3	9–1.06 K
Expo4D2M	2 M	4	23–9.26 K	*Expo4D10M*	10 M	4	3–1.63 K
Expo6D2M	2 M	6	0–2.68 K	*Expo6D10M*	10 M	6	0–499
Expo8D2M	2 M	8	0–157	*Expo8D10M*	10 M	8	0–167
SW2DA	1.86 M	2	295–5.82 K	*SW2DB*	5.16 M	2	91–2.03 K
SW3DA	1.86 M	3	239–13.20 K	*SW3DB*	5.16 M	3	33–2.13 K
Gaia	50 M	2	19–455	*OSM*	50 M	2	67–571
SDSS	15.23 M	2 2	1–31				

The *Expo-* datasets are exponentially distributed synthetic datasets (using $\lambda = 40$), while the others are real-world datasets

- *OSM* [37], which is a collection of GPS point data from OpenStreetMap, and in which we also select 50M objects.

In addition to the real-world datasets, we conduct experiments on exponentially distributed synthetic datasets made of 2M and 10M points spanning two to eight dimensions. (We detail later the datasets used to evaluate the static partitioning in particular.) These datasets are named using the dimensions and number of points; for example, *Expo3D2M* is a three-dimensional dataset containing 2M points. We elect to use an exponential distribution (with $\lambda = 40$) as this distribution contains over-dense and under-dense regions, similarly to the real-world datasets we select. Finally, exponential distributions yield a high load imbalance between the points and thus should illustrate the performance of HEGJoin when there is a load imbalance between the CPU and GPU. Furthermore, we do not use uniformly distributed datasets, as this type of dataset would not yield load imbalance, as all the query points would have roughly the same workload. We summarize these real-world and exponentially distributed synthetic datasets in Table 2.

5.3 Methodology

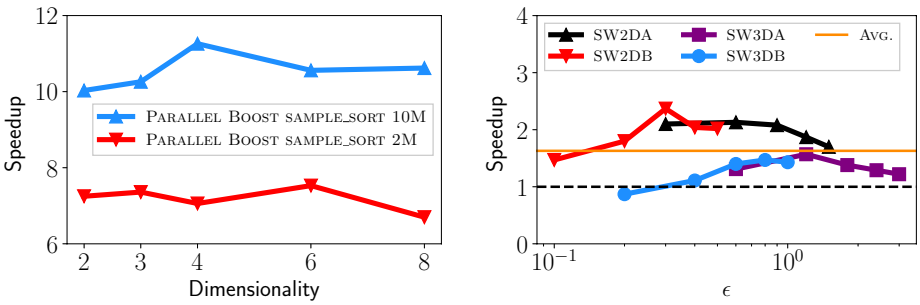
We conduct all our experiments on the two following platforms:

- *Platform 1* 2 × Intel Xeon E5-2683-v4 (with 2 × 16 cores), 256 GiB of main memory, and an Nvidia Titan X with 12 GiB of global memory.
- *Platform 2* 2 × Intel Xeon E5-2620-v4 (with 2 × 8 cores), 128 GiB of main memory, and an Nvidia Quadro GP100 with 16 GiB of global memory.

While we systematically present the results of the experiments using Platform 1, we only show the results of the experiments using Platform 2 in Fig. 13. The code executed by the CPU is written in 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 evaluate 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.

- LBJoin: the GPU algorithm proposed by [6], using 3 GPU streams (managed by 3 CPU threads), 256 threads per block, $n_s = 5 \times 10^7$ key/value pairs, where the dataset is stored as 64-bit floats, and n_p^{GPU} is given by the batch estimator presented in Sect. 4.3. This configuration is used on both platforms.
- Super-EGO: the CPU algorithm developed by [3], using 32 (16) CPU threads on Platform 1 (Platform 2), and we use 32-bit floats to store the dataset.
- New-Super-EGO: our optimized version of Super-EGO as presented in Sect. 4.5 that uses the sorting by workload strategy, using 32 (16) CPU threads on Platform 1 (Platform 2), and where the dataset is stored as 64-bit floats.
- HEGJoin-GPU: the GPU component of HEGJoin, using the same configuration as LBJoin and one of the work partitioning strategies we propose (Sect. 4.2).
- HEGJoin-CPU: the CPU component of HEGJoin, using the same configuration as New-Super-EGO and one of the work partitioning strategies (with $n_p^{\text{CPU}} = 1,024$ when using the dynamic partitioning and the shared deque).
- HEGJoin: the heterogeneous algorithm that combines HEGJoin-CPU and HEGJoin-GPU. HEGJoin-Dyn



(a) EGO-Sort Speedup **(b)** New-Super-EGO Speedup

Fig. 7 **a** Speedup to EGO-Sort our exponentially distributed 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 New-Super-EGO over Super-EGO on the *SW*- real-world datasets. Results from Platform 1

denotes HEGJoin when using the dynamic partitioning strategy, HEGJoin-SQ denotes HEGJoin when using the static partitioning strategy based on query points, while HEGJoin-SC denotes HEGJoin when using the static partitioning strategy based on candidate points.

HEGJoin-CPU and HEGJoin-GPU, as part of HEGJoin, each compute a fraction of the work. LBJoin, Super-EGO, New-Super-EGO and HEGJoin are standalone algorithms, and thus compute all the work. All response times are averaged over three time 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).

5.4 Results

In this section, we present the results of our experimental evaluation. We provide a roadmap for the organization of our results as follows:

- In Sect. 5.4.1, we present the performance of New-Super-EGO when compared to Super-EGO.
- In Sect. 5.4.2, we compare the search space pruning efficiency of the indexes used by New-Super-EGO and LBJoin.
- In Sect. 5.4.3, we outline the accuracy of the models used by HEGJoin-SQ and HEGJoin-SC that we proposed in Sects. 4.2.2 and 4.2.3.
- After the baseline performance is demonstrated in Sects. 5.4.1–5.4.3, in Sect. 5.4.4 we show the performance of HEGJoin-Dyn, HEGJoin-SQ and HEGJoin-SC, as compared to the leveraged algorithms, New-Super-EGO and LBJoin.
- In Sect. 5.4.5, we evaluate the efficiency of our shared work queue by measuring the load imbalance between the CPU and GPU.
- In Sect. 5.4.6, we assess the overhead incurred by the data transfers between the CPU and GPU when using HEGJoin.

5.4.1 Performance of New-Super-EGO

In this section, we evaluate the performance of New-Super-EGO, the optimized version of Super-EGO. The major optimizations include a different sorting algorithm, using the sorting by workload strategy and work queue (Sect. 4.5). The experiments in this section were conducted on a selection of datasets from Table 2. The results we show in this section are from using Platform 1.

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 New-Super-EGO (and thus by HEGJoin), while qsort is used by Super-EGO. Figure 7a 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 7b plots the speedup of New-Super-EGO over Super-EGO on the *SW*- real-world datasets. New-Super-EGO achieves an average speedup of 1.63 × over Super-EGO. While New-Super-EGO stores data as 64-bit floats, Super-EGO only uses 32-bit floats and thus has a performance advantage compared to New-Super-EGO. The overall speedup is explained by using sample_sort over qsort, and the sorting by workload strategy with the work queue. Therefore, New-Super-EGO largely benefits from balancing the workload between its threads and from using the work queue.

Table 3 Comparison of the number of candidate points refined by LBJoin vs. New-Super-EGO, and ratio of the number of candidate points refined by LBJoin over the number of candidate points refined by New-Super-EGO on a selection of our datasets (Table 2)

Dataset	ϵ	S	LBJoin	New-Super-EGO	Ratio
SW2DA	1.5	5.82 K	28,441,701,752	27,786,778,388	1.02
SDSS	0.002	31	65,531,735,119	66,154,801,616	0.99
SW3DA	3.0	13.20 K	90,349,946,258	87,855,196,567	1.03
Expo2D2M	0.002	9.39 K	51,789,286,408	50,121,273,123	1.03
Expo2D10M	0.0004	1.99 K	56,439,981,246	54,645,837,741	1.03
Expo4D2M	0.01	9.26 K	113,929,159,776	177,787,029,288	0.64
Expo4D10M	0.004	1.63 K	217,420,698,818	216,050,585,244	1.01
Expo8D2M	0.015	157	77,827,299,052	108,430,322,625	0.72
Expo8D10M	0.012	167	207,650,110,734	374,000,045,202	0.56

Results from Platform 1

5.4.2 Candidate Point Pruning Efficiency of LBJoin and New-Super-EGO

In this section, we explore the pruning efficiency of the grid when used by LBJoin and when used by New-Super-EGO. As we mentioned in Sect. 4.2.3, because LBJoin and New-Super-EGO use two different grid indexes, the pruning of the search space may yield a different number of candidate points to refine. Hence, we compare in Table 3 the number of candidate points refined by LBJoin and New-Super-EGO, as well as the ratio of the number of candidate points refined by LBJoin over the number of candidate points refined by New-Super-EGO on a selection of datasets. We observe that in lower dimensions, the difference in the number of candidate points refined by LBJoin and New-Super-EGO is relatively low, as the ratio is around 1. However, as dimensionality increases, we observe that this ratio tends to decrease, indicating that New-Super-EGO becomes less efficient at pruning the search space than LBJoin. The results we show in this section are from Platform 1.

5.4.3 Model Validation for HEGJoin-SQ and HEGJoin-SC

In this section, we evaluate the accuracy of the models we propose for the static partitioning strategies based on query points (Sect. 4.2.2) and based on the number of candidate points to refine (Sect. 4.2.3). The results we show in this section are from Platform 1.

Figure 8 plots the modeled execution time of LBJoin as T^{GPU} and the modeled execution time of New-Super-EGO as T^{CPU} on a selection of datasets. We observe that in 2D (Fig. 8a and b), the model determines an execution time similar to the execution time of LBJoin and New-Super-EGO. In 4D (Fig. 8c), while the modeled time for LBJoin is accurate, the modeled time of New-Super-EGO is overestimated when $\epsilon > 2.4 \times 10^{-3}$. On the *Expo6D10M* dataset (Fig. 8d), we observe that the modeled time of both LBJoin and New-Super-EGO are overestimated when $\epsilon > 4.8 \times 10^{-3}$. Thus, we observe that the model may sometimes not accurately predict the response time of HEGJoin and, therefore, may yield a poor distribution of the work to the CPU and GPU.

This poor distribution is particularly impactful when the execution time of a processor is overestimated, while the execution time of the other processor is underestimated. As the model yields $f_q = f_c$, the workload of the static partitioning based on the query points is likely to be higher than the workload of the static partitioning based on the number of candidate points to refine. Indeed, because the query points are sorted by their workload in a non-increasing order (Sect. 3.1.3), the number of query points determined by the

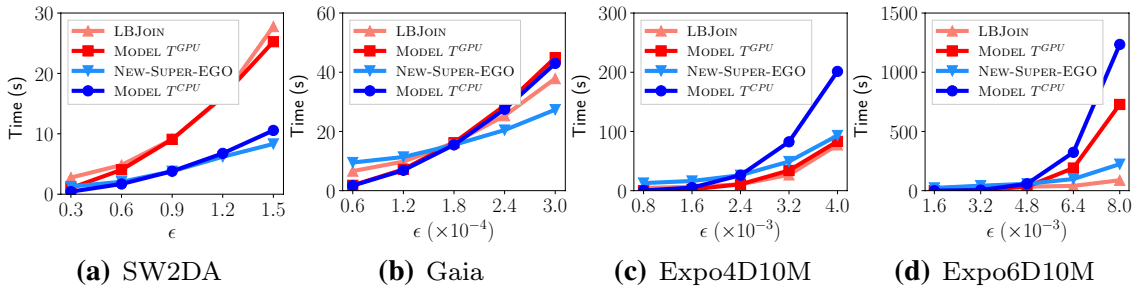


Fig. 8 Comparison of the modeled execution times T^{GPU} and T^{CPU} vs. their corresponding reference execution times LBJoin and New-Super-EGO on a selection of datasets: (a) *SW2DA*, (b) *Gaia*, (c) *Expo4D10M* and (d) *Expo6D10M*. The results from Platform 1

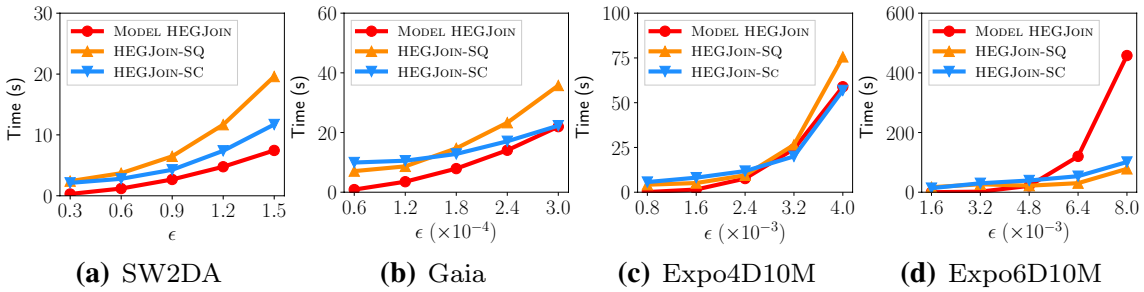


Fig. 9 Comparison of the modeled execution time of HEGJoin as determined by the static partitioning model vs. the response time of HEGJoin-SQ and HEGJoin-SC on a selection of datasets: **a** *SW2DA*, **b** *Gaia*, **c** *Expo4D10M* and **d** *Expo6D10M*. Results from Platform 1

static partitioning fraction f_q will very likely have a cumulative workload higher than the workload yielded by the static partitioning of the candidate points determined by the static partitioning fraction f_c.

Figure 9 plots the modeled execution time of HEGJoin as determined by the static partitioning model (Sect. 4.2.2) vs. the response time of HEGJoin-SQ and HEGJoin-SC on a selection of datasets. We observe on the 2D datasets (Fig. 9a and b) that the modeled execution time for HEGJoin is slightly underestimated compared to the execution time of HEGJoin-SQ and HEGJoin-SQ. As we mentioned before, low-dimensional searches are memory-bound, a bottleneck that the model is unable to capture and thus to include in its modeled time. Indeed, as we consider the upper bound throughput as the sum of LBJoin and New-Super-EGO respective throughput, we assume that concurrently using the CPU and the GPU scales perfectly, without said bottlenecks. Nevertheless, the modeled execution time of HEGJoin is overall similar to the execution time of HEGJoin-SQ and HEGJoin-SC. On the *Expo4D10M*, and despite its overestimation of the modeled time of New-Super-EGO (Fig. 8c), the modeled execution time of HEGJoin is very similar to the execution time of HEGJoin-SQ and HEGJoin-SC. Finally, the overestimation of both the modeled time of LBJoin and New-Super-EGO on the *Expo6D10M* dataset (Fig. 8d) is also reflected in Fig. 9d, as the modeled execution time of HEGJoin is also overestimated compared to the execution time of HEGJoin-SQ and HEGJoin-SC. However, we observe that the modeled execution time of HEGJoin is very similar to the modeled execution time of LBJoin (Fig. 8d), which means that the model considers, for this dataset, that HEGJoin is mostly relying on the GPU to compute the majority of the work. On the *Expo6D10M* dataset, we would thus expect HEGJoin-SQ and HEGJoin-SC to have a rather high load imbalance, as the CPU is likely to have little work to compute, and therefore to have to wait for the GPU to finish its computation. We confirm this expectation in Sect. 5.4.5 when evaluating the load imbalance of the partitioning strategies we propose.

5.4.4 Performance of the Work Partitioning Strategies

In this section, we evaluate the performance of our three work partitioning strategies, i.e., HEGJoin-Dyn, HEGJoin-SQ, and HEGJoin-SC. We compare their performance to LBJoin and New-Super-EGO. While we show results for a selection of our synthetic datasets (Fig. 10) that span multiple dimensions and size, we show the results on all our real-world datasets (Fig. 11). The results we show in this section are from Platform 1.

Performance on Exponential Datasets Fig. 10 plots the response time of HEGJoin-Dyn, HEGJoin-SQ, HEGJoin-SC, LBJoin, and New-Super-EGO on the (a) *Expo2D2M*, (b) *Expo2D10M*, (c) *Expo4D10M*, (d) *Expo6D10M*, (e) *Expo8D2M*, and (f) *Expo8D10M* datasets. We select these datasets as they span multiple dimensions and different sizes. We observe on most datasets (Fig. 10a–d) that HEGJoin-Dyn and HEGJoin-SC overall yield similar performance, while HEGJoin-SQ is rather inefficient as it does not substantially improve the execution time of either LBJoin or New-Super-EGO. Thus, using the number of candidate points in the model that is used to statically partition the work yields a better work distribution than not considering the candidate points.

In Fig. 10, we observe on our highest dimensional datasets, and more particularly for the intermediate values of ϵ (*Expo8D2M* where $\epsilon = 0.9 \times 10^{-2}$, and *Expo8D10M* where $\epsilon = 0.72 \times 10^{-2}$), that HEGJoin-Dyn response time does not monotonically increase with ϵ as it is the case in lower dimensions. This occurs because few batches are executed on the GPU, and which take a significant amount of time. This prevents the CPU from taking work from the work queue, thereby increasing the load imbalance between the CPU and GPU. At higher values of ϵ, there is less load imbalance between the CPU and GPU; therefore, the response time decreases. On the other hand, we find that on these datasets (*Expo8D2M* and *Expo8D10M*), HEGJoin-SQ is often the most efficient partitioning strategy, while the performance of HEGJoin-SC is between LBJoin and

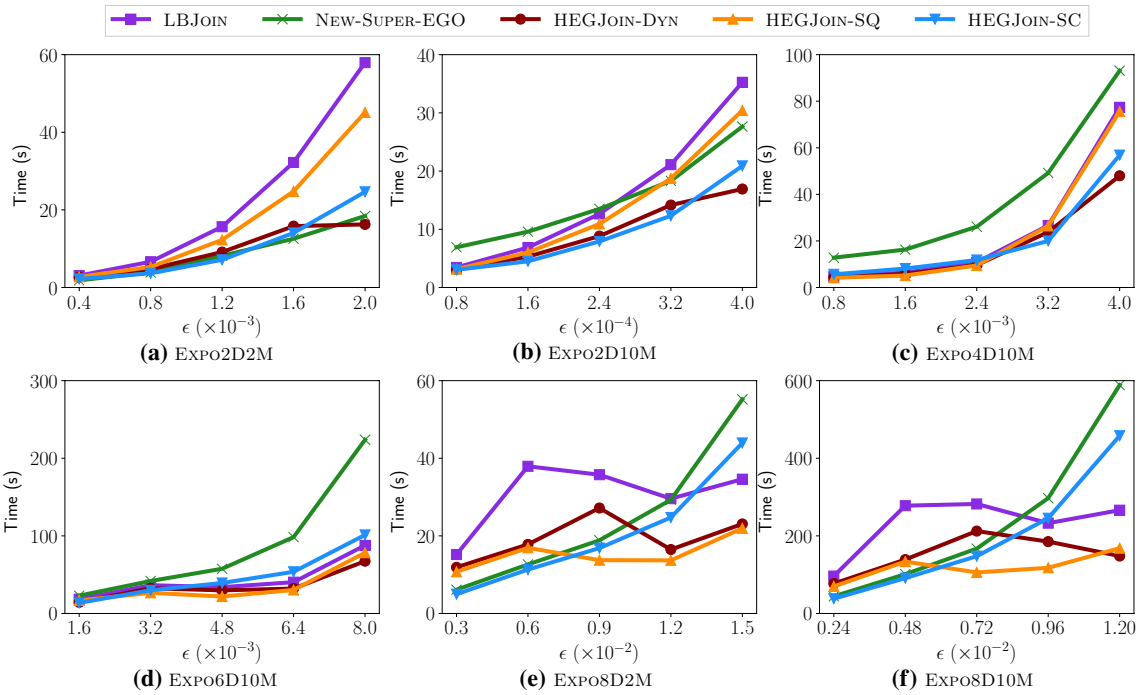


Fig. 10 Response time of HEGJoin-Dyn, HEGJoin-SQ, HEGJoin-SC, LBJoin, and New-Super-EGO on **a** *Expo2D2M*, **b** *Expo2D10M*, **c** *Expo4D10M*, **d** *Expo6D10M*, **e** *Expo8D2M*, and **f** *Expo8D10M*. *S* is in the range **a** 397–9.39 K, **b** 80–1.99 K, **c** 3–1.63 K, **d** 0–499, **e** 0–157, and **f** 0–167. Results from Platform 1

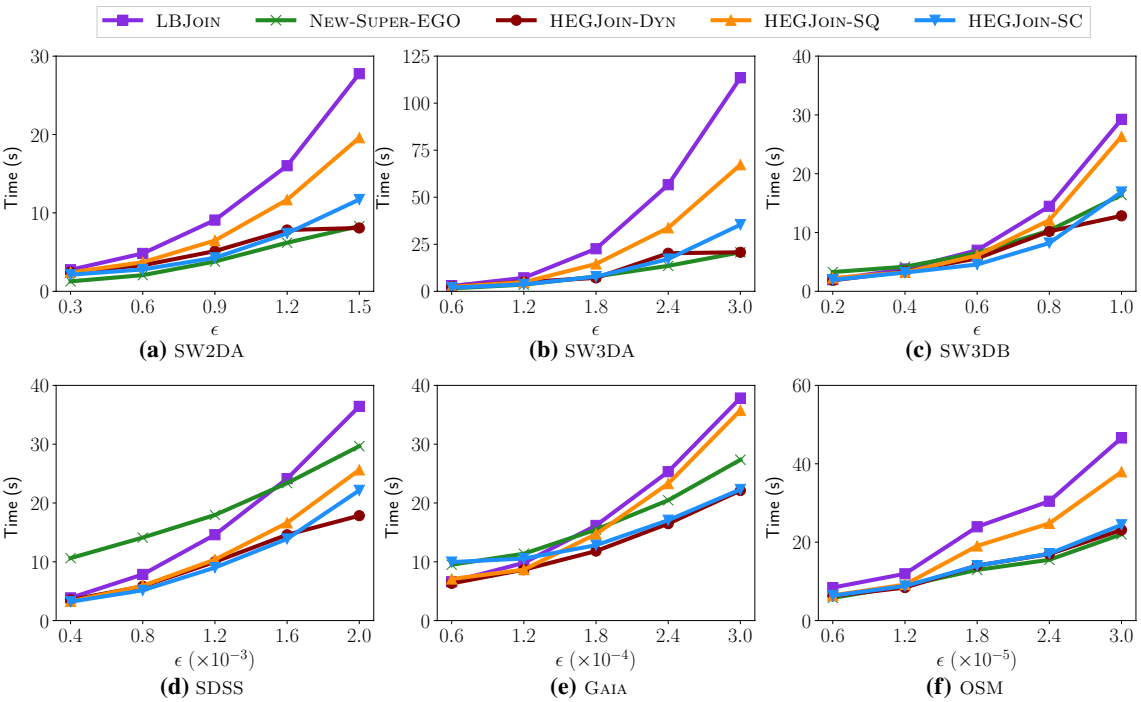


Fig. 11 Response time of HEGJoin-Dyn, HEGJoin-SQ, HEGJoin-SC, LBJoin, and New-Super-EGO on (**a**) *SW2DA*, **b** *SW3DA*, **c** *SW3DB*, **d** *SDSS*, **e** *Gaia*, and **f** *OSM*. *S* is in the range **a** 295–5.82 K, **b** 239–13.2 K, **c** 33–2.13 K, **d** 1–31, **e** 19–455, and **f** 67–571. Results from Platform 1

Table 4 Throughput of candidate points refined (candidates/s) by LBJoin, New-Super-EGO, the upper bound of LBJoin plus New-Super-EGO, HEGJoin-Dyn, and the performance ratio between HEGJoin-Dyn and the upper bound across several datasets. Results from Platform 1

Dataset	ϵ	S	LBJoin	New-Super-EGO	Upper Bound	HEGJoin-Dyn	Perf. Ratio
Expo2D2M	0.002	9392	893,877,929	2,812,071,408	3,705,949,337	3,185,072,965	0.86
Expo4D2M	0.01	9262	672,847,132	1,777,133,999	2,449,981,131	2,209,642,354	0.90
Expo8D2M	0.015	157	3,881,606,529	1,410,542,112	3,659,149,867	3,372,066,683	0.92
Expo2D10M	0.0004	1985	1,601,136,521	2,042,081,926	3,643,218,447	3,335,696,291	0.92
Expo4D10M	0.004	1630	2,809,451,506	2,334,736,697	5,144,188,204	4,531,486,011	0.88
Expo8D10M	0.012	167	2,233,156,849	1,010,004,731	3,243,161,581	4,013,791,675	1.24
SW2DA	1.5	5818	1,024,556,980	3,419,458,232	4,444,015,212	3,520,012,593	0.79
SDSS	0.002	31	1,798,770,443	2,208,414,897	4,007,185,340	3,673,303,538	0.92
Gaia	0.0003	455	1,696,613,608	2,347,964,353	4,044,577,961	2,903,111,440	0.72
OSM	0.00003	571	1,287,786,499	2,725,941,526	4,013,728,025	2,596,190,956	0.65
SW3DA	3.0	13,207	796,136,506	4,360,015,024	5,156,151,530	4,354,214,277	0.84

New-Super-EGO. If we examine LBJoin and New-Super-EGO execution times for the median value of ϵ, we observe that the CPU is more efficient than the GPU. Hence, the model assumes that the CPU is consistently more efficient for other values of ϵ and will therefore assign more work to the CPU. However, the LBJoin execution time does not increase as much as the model predicted, while New-Super-EGO execution time increased more than what the model predicted. Hence, the model will assign a higher fraction of the work to the CPU than it is capable of processing within the execution time estimated by the model. On these particular datasets (*Expo8D2M* and *Expo8D10M*), since the execution time of LBJoin is overestimated and the execution time of New-Super-EGO is underestimated, the static partitioning based on query points ends up being the most efficient partitioning as ϵ increases, since most of the work is assigned to the GPU.

As described in Section 1, we choose to focus on low dimensionality. Observe here that the execution time of New-Super-EGO significantly degrades with dimensionality (Fig. 10d–f). Therefore, if we were to employ New-Super-EGO at higher dimensions than that explored in this work, the algorithm would have a negligible impact on performance of HEGJoin. In higher dimensions, it would be more worthwhile to consider the use of a different CPU algorithm to replace New-Super-EGO, such as that proposed by [33].

Performance on Real-World Datasets Fig. 11 plots the response time of HEGJoin-Dyn, HEGJoin-SQ, HEGJoin-SC, LBJoin, and New-Super-EGO on the (a) *SW2DA*, (b) *SW3DA*, (c) *SW3DB*, (d) *SDSS*, (e) *Gaia*, and (f) *OSM* datasets. We observe on these real-world datasets a similar behavior as on the *Expo2D2M* and *Expo2D10M* datasets (Fig. 10a and b). Thus, we observe that the static partitioning based on the number of candidate points to refine, HEGJoin-SC, achieves similar performance as the dynamic partitioning HEGJoin-Dyn. Furthermore, we can see that both of these partitioning strategies achieve similar or better performance than the best performance yielded by LBJoin or New-Super-EGO. Furthermore, we observe that HEGJoin-SQ yields poor performance. As we explained in Section 5.4.3, because the execution time may be overestimated or underestimated, a processor can be assigned too much work or too little work relative to its real computational throughput.

Candidate Point Refinement Throughput Table 4 presents the candidate point refinement throughput (as previously defined in Sect. 4.2.3) for LBJoin, New-Super-EGO, HEGJoin-Dyn, the upper bound (the total throughput given by adding the throughput of the standalone LBJoin and New-Super-EGO algorithms), and the ratio of the throughput HEGJoin-Dyn achieves compared to this upper bound throughput. The candidate throughput corresponds to the number of candidate points to refine, divided by the response time of the algorithm, as shown in Figs. 10 and 11. We observe a relatively high performance ratio, demonstrating that we almost reach the performance upper bound of HEGJoin. Moreover, we also observe that on the *Expo8D10M* dataset, we achieve a ratio of more than 1. We explain this by the fact that *Expo8D10M* is exponentially distributed and therefore has very dense regions, as well as very sparse regions. Thus, the throughput of LBJoin includes query points with a very low workload, thus increasing its overall throughput compared to what HEGJoin-GPU achieves. Similarly, the throughput of New-Super-EGO 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

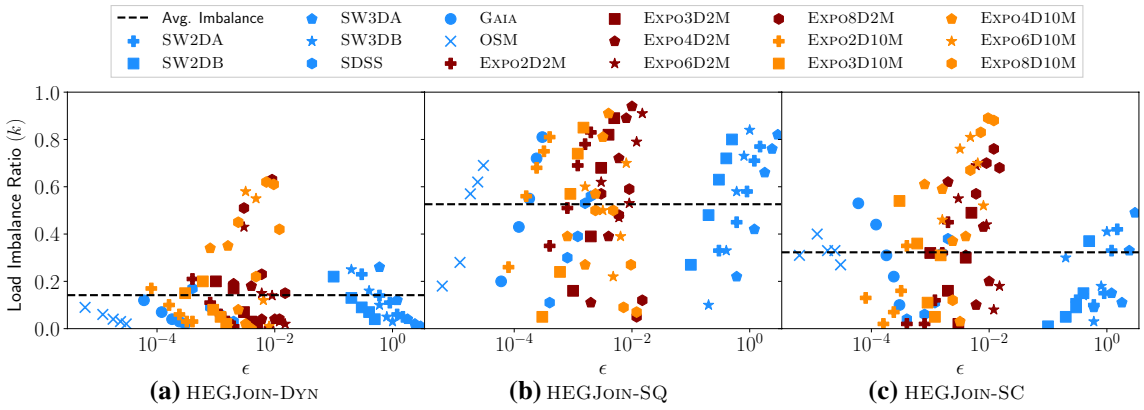


Fig. 12 Load imbalance ratio of (**a**) HEGJoin-Dyn, **b** HEGJoin-SQ and **c** HEGJoin-SC on all the datasets we use for our experiments, and we described in Table 2. The horizontal dashed line corresponds to the average load imbalance k, and is as follows: **a** $k = 0.14$, **b** $k = 0.53$ and **c** $k = 0.32$. Results from Platform 1

for the GPU and higher for the CPU, than their throughput when computing the entire dataset.

Performance ratios lower than 1 (Table 4) indicate that there are several bottlenecks, including contention for memory bandwidth, with the peak bandwidth potentially reached when concurrently 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 higher memory pressure than in higher dimensions or for higher selectivity (Figs. 10 and 11). We confirm this by examining the ratio of kernel execution time over the time to compute all batches of LBJoin. Focusing on the datasets with the minimum and maximum performance ratio from Table 4, we find that on *Gaia*, LBJoin has a kernel execution time ratio of 0.06, while on *Expo8D10M*, LBJoin has a kernel execution time ratio of 0.98. Hence, most of the *Gaia* execution time is spent on memory operations, while on *Expo8D10M*, the execution time is mostly spent on computation. When executing LBJoin on *Gaia* (and other datasets with low ratios in Table 4), we observe that the use of the GPU hinders the CPU by using a non-negligible fraction of the total available memory bandwidth.

5.4.5 Load Balancing Efficiency

We define the load imbalance of HEGJoin as follows. Given the total execution time T, the time t^{GPU} (t^{CPU}) at which the GPU (CPU) ends its work, we characterize the load imbalance ratio as $k = (|t^{\text{GPU}} - t^{\text{CPU}}|)/T$. A load imbalance ratio close to 0 therefore indicates that the CPU and GPU ended their work at roughly the same time, and thus, that the load imbalance between the CPU and GPU is low. The results we show in this section are from Platform 1.

Figure 12 plots the load imbalance ratio of (a) HEGJoin-Dyn, (b) HEGJoin-SQ and (c) HEGJoin-SC across all the datasets we present in Table 2. We observe in Fig. 12a that HEGJoin-Dyn (Sect. 4.2.1) achieves a fairly good load balancing, as it achieves an average load imbalance ratio of $k = 0.14$. Furthermore, the datasets in higher dimensions (such as *Expo6D-* and *Expo8D-*) are distinguished by a high load imbalance (with the highest load imbalance ratio, $k = 0.62$, recorded on the *Expo8D10M* dataset and for $\epsilon = 0.72 \times 10^{-2}$). We explain this by the fact that the computation on these datasets is made in only a few large batches (Sect. 3.1.2) and thus explained by the CPU and GPU less frequently accessing the shared deque than in lower dimensions. While having more batches with a reduced size would improve load balancing, it would negatively impact the GPU's performance, as the GPU may be underutilized.

Figure 12b plots the load imbalance ratio of the static partitioning based on query points, HEGJoin-SQ (Sect. 4.2.2). We immediately observe a high average load imbalance of $k = 0.53$, meaning that on average, the CPU or GPU spend half of the execution time idle. Hence, HEGJoin-SQ yields a load imbalance of up to $k = 0.91$ on the *Expo4D10M* dataset when $\epsilon = 4.0 \times 10^{-3}$. Considering that HEGJoin-SC is usually more efficient than HEGJoin-SQ (Figs. 10 and 11) and yet uses the same model to predict the execution time, the high load imbalance of HEGJoin-SQ is therefore explained by how the work is partitioned between the CPU and GPU, based on query points. Indeed, as we explained above (Sect. 5.4.4), on datasets such as *Expo8D2M* and *Expo8D10M* (Fig. 10e and f), as the query points are sorted by workload, the workload assigned to the GPU when partitioning based on query points is higher than the workload assigned when partitioning based on the number of candidate points to refine. Examining the average load imbalance

Table 5 Total time taken by data transfers between the CPU and the GPU, the total execution time of the algorithm, and the upper bound overhead ratio of the data transfers time to the execution time when using HEGJoin-Dyn

Dataset	ϵ	S	Data transfer time (s)	Execution time (s)	Upper bound overhead ratio
Expo2D2M	0.002	9342	5.11	30.99	0.16
Expo4D2M	0.01	9262	9.22	81.97	0.11
Expo8D2M	0.015	157	0.22	18.47	0.01
Expo2D10M	0.0004	1985	6.69	27.16	0.25
Expo4D10M	0.004	1630	8.21	55.27	0.15
Expo8D10M	0.012	167	1.19	116.61	0.01
SW2DA	1.5	5818	2.66	13.91	0.19
SDSS	0.002	31	7.72	29.74	0.26
Gaia	0.0003	455	9.77	37.83	0.26
OSM	0.00003	571	8.73	36.98	0.24
SW3DA	3.0	13,207	9.41	40.73	0.23

A ratio close to zero indicates an insignificant overhead incurred by data transfers compared to the total execution time. The ratios do not account for the periods of time where the data transfers and the kernel executions overlap. The times were recorded on the Nvidia Visual Profiler over a single time trial using Platform 2

across all values of ϵ of the *SW2DA* dataset, we observe that the average load imbalance is $k = 0.12$ for HEGJoin-Dyn, $k = 0.22$ for HEGJoin-SC, and $k = 0.57$ for HEGJoin-SQ. However, on the *Expo8D10M* dataset and for all the values of ϵ we experiment on this dataset with, the average load imbalance is $k = 0.46$ for HEGJoin-Dyn, $k = 0.66$ for HEGJoin-SC, while $k = 0.29$ for HEGJoin-SQ. Hence, the situations where HEGJoin-SQ achieves a low load imbalance are essentially exceptions to the rather bad performance of HEGJoin-SQ, as they are the result of the model's inaccuracy in such situations.

Figure 12c plots the load imbalance ratio of HEGJoin-SC, i.e., HEGJoin using the static partitioning based on the number of candidate points to refine (Sect. 4.2.3). We observe an average load imbalance ratio of $k = 0.32$, which is between the average load imbalance ratio of HEGJoin ($k = 0.14$) and HEGJoin-SQ ($k = 0.53$). Furthermore, the highest load imbalance yielded by this static partitioning strategy is on the *Expo8D10M* dataset when $\epsilon = 0.96 \times 10^{-2}$, where $k = 0.89$. The issue on this dataset is the same as when partitioning based on query points: the model is not able to predict situations where the execution time of one of the processors does not increase as the model predicts it will (in this case the execution time of LBJoin). Considering that the execution time of LBJoin is overestimated and that the execution time of New-Super-EGO is underestimated, the GPU is assigned a lower workload and the CPU a higher workload than what they are able to process within the modeled execution time. Finally, and despite HEGJoin-SC having a higher load imbalance than HEGJoin-Dyn, we observe that HEGJoin-SC is roughly as efficient as HEGJoin-Dyn on many datasets and values of ϵ.

5.4.6 Data Transfer Overhead of HEGJoin

In this section, we evaluate the overhead of the data transfers between the CPU's main memory and the GPU's global memory, regardless of their direction (from the CPU to the GPU, and vice versa), which is known to be a bottleneck due to the relatively low memory bandwidth of the PCIe-3 interconnect [38]. In Table 5, we report the time taken by all the data transfers between the CPU and GPU, the total execution time, and the ratio of the data transfers time to the total execution time, across a selection of datasets and ϵ values. The values are recorded when using HEGJoin-Dyn on Platform 2, over a single trial, and are measured using the Nvidia Visual Profiler. A ratio close to zero indicates that the data transfers are negligible relative to the total execution time of the algorithm, while higher ratios account for a large fraction of the total execution time and thus may degrade performance. Recall that we use three streams to overlap data transfers with computation (Sect. 3.1.2). However, since there is not a direct way to account for the overlap of data transfers with kernel execution (computation), we consider here that no such overlap occurs. Therefore, ratios we report in Table 5 capture the upper bound (worst-case) data transfer overhead.

We observe in Table 5 that the ratios of the data transfers time to the total execution time of HEGJoin-Dyn are relatively low across our experiments, despite only capturing the upper bound as described above. Furthermore, the experiments with the highest selectivity (e.g., *SW3DA* for $\epsilon = 3.0$) or the largest datasets (e.g., *Gaia*) yield the highest overhead ratios, due to the large result set size that must be transferred from the GPU to the CPU, or large datasets that must be transferred from the CPU to the

GPU. However, in these cases, the relatively high selectivity also yields a large number of batches to compute, making it easier to overlap data transfers with kernel executions, which we could not account for here. On experiments with a lower selectivity (e.g., *Expo8D10M* when $\epsilon = 0.012$), the data transfers account for an insignificant amount of time compared to the high total execution time of HEGJoin-Dyn, due to the relatively small size of the result set that needs to be transferred from the GPU to the CPU. Overall, and given that the overhead ratios in Table 5 consist of an upper bound, we consider that the data transfers between the CPU and the GPU are marginally impacting the performance of HEGJoin.

5.5 Discussion

We summarize and discuss the major research findings in this paper. We find that HEGJoin using the on-demand work queue (HEGJoin-Dyn) outperforms the two static partitioning methods (HEGJoin-SQ and HEGJoin-SC) on most values of ϵ. Despite this finding, HEGJoin-Dyn does not achieve low load imbalance between the CPU and GPU components of the algorithm across all experimental scenarios (e.g., in Fig. 12 there is a mean load imbalance of $k = 0.14$). Therefore, dynamically assigning work to the CPU and GPU components of the algorithm is challenging, even when distributing the work on-demand. Part of the reason load imbalance occurs is because the performance characteristics are fundamentally data-dependent regardless of the self-join algorithm (assuming such an algorithm uses the search-and-refine strategy). Query points have differing amounts of work to compute, so it is difficult to split the work and obtain low load imbalance between the CPU and GPU regardless of the method used to distribute the work.

Regarding our performance models, we find that individually modeling the response time of New-Super-EGO and LBJoin is accurate in some cases, and inaccurate in others (Fig. 8). We constrained the model to only require a single time measurement of New-Super-EGO and LBJoin on each dataset. This restriction means that if the response time increases nonlinearly as a function of the search volume, then the model is unable to adequately capture the measured response time. This led to the models overestimating the response time in some cases, yielding a poor distribution of work between the CPU and GPU for the static splitting strategies (HEGJoin-SQ and HEGJoin-SC).

Our static partitioning strategies that distribute the work based on the performance models considered: (*i*) all query points have an identical amount of work to compute (HEGJoin-SQ), and (*ii*) query points have a varying amount of work to compute based on the size of each query point's candidate set (HEGJoin-SC). A good distribution of work to the CPU and GPU requires that the models are able to adequately capture performance, and we demonstrated this by showing that the partitioning strategy based on (*ii*) outperforms (*i*) above. Despite HEGJoin-SC being able to capture the number of candidate points that need to be refined per query point, we find that the model was unable to capture several performance characteristics that degraded the performance of this static partitioning strategy in some cases. We outline some factors that contribute to poor model accuracy as follows.

1. The size of the GPU batches must be substantially larger to saturate GPU resources; therefore, this increases the chances that the CPU will be starved of work toward the end of the computation, leading to non-negligible load imbalance.
2. The GPU component of HEGJoin reduces the main memory bandwidth of the CPU component; therefore, if ϵ and data properties lead to a memory-bound execution, the GPU's memory operations will reduce the CPU's available memory bandwidth, which will lead to load imbalance.
3. Depending on data properties and ϵ, the GPU may be underutilized due to many factors, including those related to the SIMT architecture. Typically, this occurs when we observe that the response time is roughly "flat" with increasing ϵ (Fig. 10e and f). Since the GPU may be underutilized, increasing ϵ has little impact on performance, which causes the model to overestimate the response time. An example of this is shown in Fig. 9d by comparing the execution time of HEGJoin-SC and model curves.
4. Algorithms for the CPU typically achieve the best performance (lowest response time) if they are work-efficient. However, the GPU's architecture can break this work-efficient assumption, as algorithms designed for the GPU may be work-inefficient but achieve a lower response time than a work-efficient algorithm that performs the same task [39]. Consequently, modeling the performance of a GPU-only algorithm is challenging, and the addition of a concurrently executing CPU algorithm exacerbates this problem.

In summary, this paper yields insight into the self-join as executed on heterogeneous architectures, which necessitates a comprehensive examination of the problem of work distribution between architectures. The insights described above outline several challenges related to splitting the work using static and dynamic partitioning strategies. Despite these aforementioned challenges, HEGJoin is more robust to dataset characteristics and search distance, as we find that the algorithm generally outperforms the CPU/GPU-only counterparts. We show the speedup of HEGJoin

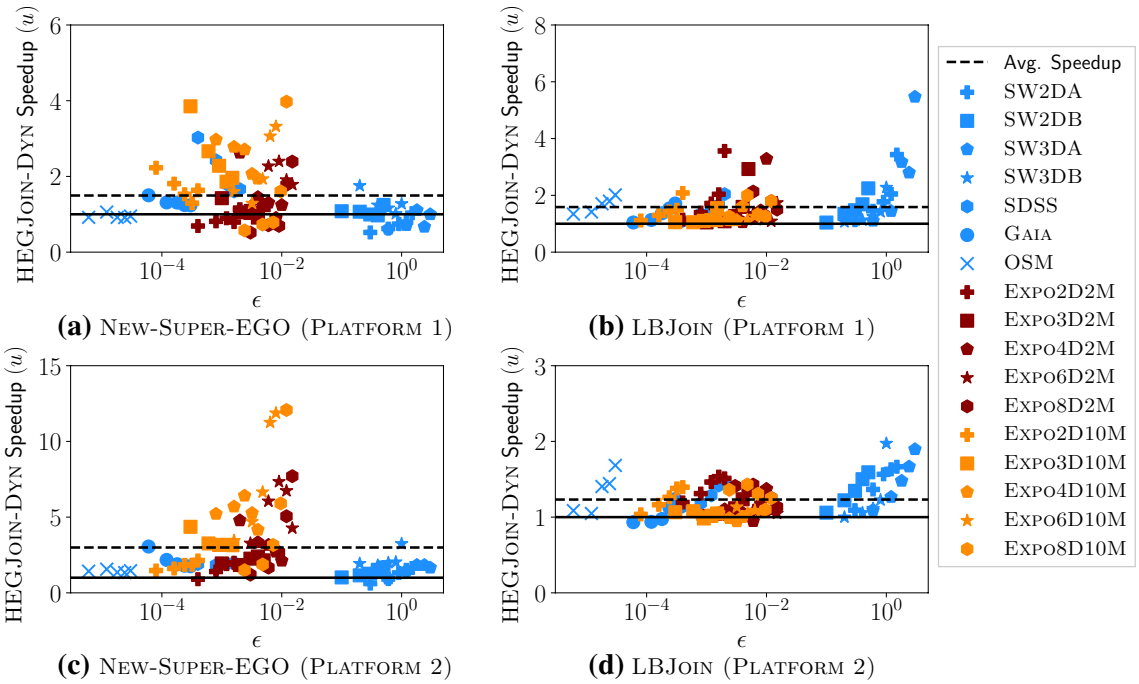


Fig. 13 Average speedup of HEGJoin-Dyn over **a** New-Super-EGO and **b** LBJoin on Platform 1, and over **c** New-Super-EGO and **d** LBJoin on Platform 2, across all datasets (Table 2). The horizontal dashed line corresponds to the average speedup u and is as follows: **a** $u = 1.50$, **b** $u = 1.59$, **c** $u = 2.99$, and **d** $u = 1.23$. The horizontal solid line corresponds to a speedup of $u = 1.0$

over New-Super-EGO and LBJoin across all our datasets (Table 2) and evaluated on two different platforms in Fig. 13. We thus observe that HEGJoin-Dyn is, independently from the platform we used, on average more efficient than LBJoin and New-Super-EGO.

6 Conclusion

In this paper, we propose HEGJoin, which is to the best of our knowledge the first data-parallel heterogeneous and concurrent CPU-GPU algorithm that computes distance similarity searches and that leverages LBJoin and Super-EGO, two state-of-the-art algorithms to compute distance similarity searches on the GPU and CPU, respectively. While the computation of distance similarity searches is memory-bound in lower dimensions, it becomes compute-bound in higher dimensions. In both of these situations, the GPU is very suitable at computing distance similarity searches, due to its higher computational throughput and memory bandwidth compared to the CPU.

We propose three work partitioning strategies to assign work to the CPU and GPU; particularly, we propose a dynamic work partitioning strategy that assigns work to the CPU and GPU on-demand through a shared deque, in addition to two static partitioning strategies based on the number of query points, and based on the number of candidate points that will need to be refined. The dynamic partitioning strategy simply does not consider the overall workload of HEGJoin and is efficient because of the shared deque and its on-demand work assignment to the CPU and GPU. In contrast, the static partitioning strategy HEGJoin-SQ is workload-oblivious, while HEGJoin-SC is workload-aware.

We described several insights into the work partitioning problem between the CPU and GPU based on the static partitioning strategies. To summarize, the use of two different architectures, combined with two different algorithms makes modeling HEGJoin a challenging task. This led to dynamic partitioning being generally more efficient than the two static partitioning strategies. Despite the challenges of statically partitioning the work, we find that HEGJoin with the dynamic deque is more robust to data distributions and the search radius of the self-join than the CPU-only and GPU-only algorithms. Consequently, HEGJoin outperforms the CPU/GPU-only algorithms in most experimental scenarios.

The dynamic partitioning strategy achieved the best performance. Future work should examine different ways to enhance the dynamic partitioning method described in this paper, while still being able to accommodate the GPU's requirement of processing large batches of work to achieve high search throughput. By narrowing our focus on this task, we may be able to further reduce the load imbalance observed, particularly on higher-dimensional datasets. Another research direction is to use nonparametric models

to statically split the work between the CPU and GPU, which may be able to better capture the complexity of the algorithm. Another possibility is to use an adaptive model that could systematically select the best work partitioning strategy based on data characteristics.

Acknowledgements This material is based upon work supported by the National Science Foundation under Grant No. 1849559. We thank Frédéric Loulergue for letting us use his platform to conduct our experiments.

References

1. Böhm C, Braunmüller B, Breunig MM, Kriegel H-P (2000) High Performance clustering based on the similarity join. In: Proceedings of the international conference on information and knowledge management, pp 298–305
2. Böhm C, Braunmüller B, Krebs F, Kriegel H-P (2001) Epsilon grid order: An algorithm for the similarity join on massive high-dimensional data. In: Proceedings of the ACM SIGMOD international conference on management of data, pp 379–388. ISBN 1-58113-332-4
3. Kalashnikov DV (2013) Super-EGO: fast multi-dimensional similarity Join. VLDB J 22(4):561–585
4. Böhm C, Noll R, Plant C, Zherdin An (2009) Index-supported similarity join on graphics processors, pp 57–66
5. Lieberman MD, Sankaranarayanan J, Samet H (2008) A fast similarity join algorithm using graphics processing units. In: 2008 IEEE 24th international conference on data engineering, pp 1111–1120
6. Gallet B, Gowanlock M (2020) Load imbalance mitigation optimizations for GPU-accelerated similarity joins. In: Proceedings of the 2019 IEEE international parallel and distributed processing symposium workshops, pp 396–405, https://github.com/benoitgallet/self-join-hpbdc2019 (Accessed 17 June 2019)
7. Liao S, Lopez MA, Leutenegger ST (2001) High dimensional similarity search with space filling curves. In: Proceedings 17th international conference on data engineering, pp 615–622
8. Razenshteyn I, Schmidt L, Andoni A, Indyk P, Laarhoven T (2020) FALCONN: similarity search over high-dimensional data, 2015–2016. https://falconn-lib.org/ (Accessed 17 June 2019)
9. Bellman RE (1961) Adaptive control processes: a guided tour. Princeton University Press, Princeton
10. Awad MA, Ashkiani S, Johnson R, Farach-Colton M, Owens JD (2019) Engineering a High-performance GPU B-Tree. In: Proceedings of the 24th symposium on principles and practice of parallel programming, pp 145–157. ISBN 978-1-4503-6225-2
11. Yan Z, Lin Y, Peng L, Zhang W (2019) Harmonia: a high throughput B+Tree for GPUs. In: Proceedings of the 24th symposium on principles and practice of parallel programming, pp 133–144. ISBN 978-1-4503-6225-2
12. Jinwoong K, Sul-Gi K, Beomseok N (2013) Parallel multi-dimensional range query processing with R-trees on GPU. J Parallel Distrib Comput 73(8):1195–1207 ISSN 07437315
13. Kim J, Jeong W, Nam B (2015) Exploiting massive parallelism for indexing multi-dimensional datasets on the GPU. IEEE Trans Parallel Distrib Syst 26(8):2258–2271 ISSN 1045-9219
14. Prasad SK, McDermott M, He X, Puri S (2015) GPU-based parallel R-tree construction and querying. In: 2015 IEEE international parallel and distributed processing symposium workshops, pp 618–627
15. Jinwoong K, Beomseok N (2018) Co-processing heterogeneous parallel index for multi-dimensional datasets. J Parallel Distrib Comput 113:195–203 ISSN 0743-7315
16. Gowanlock M (2019) KNN-joins using a hybrid approach: exploiting CPU/GPU workload characteristics. In: Proceedings of the 12th workshop on general purpose processing using GPUs, pp 33–42, ISBN 978-1-4503-6255-9
17. Shahvarani A, Jacobsen H-A (2016) A hybrid B+-tree as solution for in-memory indexing on CPU-GPU heterogeneous computing platforms. In: Proceedings of the international conference on management of data, pp 1523–1538. ISBN 978-1-4503-3531-7
18. Grewe D, O'Boyle MFP (2011) A static task partitioning approach for heterogeneous systems using OpenCL. In: Compiler construction, pp 286–305. ISBN 978-3-642-19861-8
19. Ogata Y, Endo T, Maruyama N, Matsuoka S (2008) An efficient, model-based CPU-GPU heterogeneous fft library. In: 2008 IEEE international symposium on parallel and distributed processing, pp 1–10
20. Ohshima S, Kise K, Katagiri T, Yuba T (2007) Parallel processing of matrix multiplication in a CPU and GPU heterogeneous environment, pp 305–318
21. NVIDIA. Nvidia Ampere Whitepaper (2020) https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/nvidia-ampere-architecture-whitepaper.pdf (Accessed 17 June 2020)
22. Nvidia (2020a) CUDA toolkit documentation: performance guidelines. https://docs.nvidia.com/cuda/index.html (Accessed 17 June 2020)
23. Nvidia (2020b) CUDA C++ programming guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html (Accessed 17 June 2020)
24. Jon Louis Bentley (1975) Multidimensional binary search trees used for associative searching. Commun ACM 18(9):509–517
25. Bayer R, McCreight EM (1972) Organization and maintenance of large ordered indexes. Acta Informatica 1(3):173–189 ISSN 1432-0525
26. Comer D (1979) The ubiquitous B-tree. ACM Comput Surv 11(2):121–137 ISSN 0360-0300
27. Guttman A (1984) R-Trees: a dynamic index structure for spatial searching. SIGMOD Rec 14(2):47–57
28. Sellis T, Roussopoulos N, Faloutsos C (1987) The R+-Tree: a dynamic index for multi-dimensional objects. In: Proceedings of the 13th VLDB conference, pp 507–518
29. Beckmann N, Kriegel H-P, Schneider R, Seeger B (1990) The R*-tree: an efficient and robust access method for points and rectangles. In: Proceedings of the ACM SIGMOD international conference on management of data, pp 322–331 . ISBN 0-89791-365-5
30. Gowanlock M, Karsin B (2019) Accelerating the similarity self-join using the GPU. J Parallel Distrib Comput 133:107–123
31. Finkel RA, Bentley JL (1974) Quad trees: a data structure for retrieval on composite keys. Acta Informatica 4(1):1–9 ISSN 1432-0525
32. Matam K, Indarapu SR, Krishna B, Kothapalli K (2012) Sparse matrix-matrix multiplication on modern architectures, pp 1–10

33. Perdacher M, Plant C, Böhm C (2019) Cache-oblivious high-performance similarity join. In: Proceedings of the 2019 international conference on management of data, SIGMOD '19, pp 87–104. ACM. https://doi.org/10.1145/3299869.3319859
34. MIT Haystack Observatory (2020) Space weather datasets. ftp://gemini.haystack.mit.edu/pub/informatics/dbscandat.zip (Accessed 17 June 2020)
35. Alam S, Albareti F, Prieto C et al (2015) The eleventh and twelfth data releases of the sloan digital sky survey: final data from SDSS-III. Astrophys J Suppl Ser 219:58
36. Gaia DR 2 (2018) https://cosmos.esa.int/web/gaia/dr2 (Accessed 17 June 2020)
37. OpenStreetMap Bulk GPS Point Data (2012) https://blog.openstreetmap.org/2012/04/01/bulk-gps-point-data/ (Accessed 17 June 2020)
38. Li A, Song SL, Chen J, Li J, Liu X, Tallent NR, Barker KJ (2020) Evaluating modern gpu interconnect: pcie, nvlink, nv-sli, nvswitch and gpudirect. IEEE Trans Parallel Distrib Syst 31(1):94–110
39. Dakkak A, Li C, Xiong J, Gelado I, Hwu W-m (2019) Accelerating reduction and scan using tensor core units. In: Proceedings of the ACM international conference on supercomputing, pp 46–57