Load Imbalance Mitigation Optimizations for GPU-Accelerated Similarity Joins

Benoît Gallet
School of Informatics, Computing and Cyber Systems
Northern Arizona University
Flagstaff, AZ, 86011
benoit.gallet@nau.edu

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

***Abstract*—The distance similarity self-join is widely used in database applications and is defined as joining a table on itself using a distance predicate. The similarity self-join is often used in spatial applications and is a building block of other algorithms, such as those used for data analysis. In this paper, we propose several new optimizations mitigating load imbalance of a GPU-accelerated self-join algorithm. The data-dependent nature of the self-join makes the algorithm potentially unsuitable for the GPU's architecture, due to variance in workloads assigned to threads. Consequently, we propose a method that reduces load imbalance and subsequent thread divergence between threads executing in a warp by considering the total workload assigned to each thread, and forcing the GPU's hardware scheduler to group threads with similar workloads within the same warp. Also, by leveraging a grid-based index, we propose a new balanced computational pattern for both reducing the number of distance calculations and the workload variance between threads. Moreover, we exploit additional parallelism by increasing the workload granularity to further improve computational throughput and workload balance within warps. Our solution achieves a speedup of up to 9.7× and 1.6× on average against another GPU algorithm, and up to 10.7× with an average of 2.5× against a CPU state-of-the-art parallel algorithm.**

***Index Terms*—GPGPU, Load balancing, Query Optimization, Range Query, Self-join**

I. Introduction

Given a dataset, the distance similarity self-join performs a range query around each point in the dataset to find each point's neighbors within a distance ϵ. The operation is used to find objects that share common properties. In databases, the self-join joins a table on itself, and this operation has been used in various fields as a building block to several algorithms such as data cleaning [1], near-duplicate detection [2], document similarity [3], or clustering algorithms [4]–[6].

Given a dataset, D, finding all the points within ϵ of a query point q is an expensive operation. A brute force implementation consisting of two nested loops has a time complexity of $O(|D|^2)$ [7]. Hence, several optimizations have been proposed to improve the performance of the distance similarity self-join, including indexing schemes, which allow pruning the search space, thus avoiding comparisons between a point and every other point in the dataset. There are hierarchical and non-hierarchical indexing schemes. The hierarchical indexing schemes are typically implemented as trees [8]–[14], while non-hierarchical structures are often implemented as grids [15]–[18]. However, as the dimensionality of the dataset increases, the efficiency of these indexes to prune the search largely diminishes. This effect is called the curse of dimensionality [19]–[21].

Several databases operations have already been optimized to be able to benefit from the most recent high-performance computing technologies such as the GPUs [12]–[14], [17], [18], [22]–[24]. Indeed, their architecture and their high-throughput make them particularly efficient at processing large volumes of data, and it is no exception for the self-join operation [17], [18], [24]. Due to their Single Instruction Multiple Threads (SIMT) architecture, GPUs are very efficient for parallel computations as threads are executed in groups called warps in CUDA terminology [25] (which we use throughout this paper). Threads in a warp are executed in parallel and in lock-step. Branching during the execution is resolved by serializing the execution of threads and their execution pathways, thus causing a loss of parallel efficiency [25]. The GPU's architecture presents several challenges concerning the distance similarity self-join. Among these issues, the total result set size may exceed global memory capacity, particularly in lower dimensions as shown in [18]. Moreover, as state-of-the-art CPU algorithms have been extensively studied, the use of a GPU may not yield a performance advantage over some of the most efficient parallel CPU algorithms.

The similarity self-join is an irregular application where each point in the dataset may not have the same number of distance calculations, depending on the data distribution. Because of the GPU's SIMT architecture, some threads with a higher workload will execute more work and thus, longer than some other threads within the same warp which will have idle periods. This issue may lead to intra-warp load imbalance and therefore decrease the throughput of the self-join. In this paper, we aim at mitigating the load imbalance of the threads within a warp, as well as between different warps. In particular, this paper makes the following contributions:

- We increase the workload granularity of each range query by assigning multiple threads to compute the distance between a query point and its potential neighbors. Having more threads computing the same query point reduces the workload imbalance within a warp, as these threads will share the same workload.

978-1-5386-5555-9/19/$31.00 ©2019 IEEE
DOI 10.1109/IPDPSW.2019.00078

- A new cell access pattern that reduces the number of redundant point comparisons. This pattern ensures that each thread assigned to a query point compares to the same number of cells, thus potentially reducing load imbalance relative to previous work.
- We reduce intra-warp load imbalance by packing points with similar workloads, quantified by the number of point comparisons, into the same warp. We order each warp in non-increasing order using their quantified total amount of work. Then, we force the GPU's hardware scheduler to execute these warps in this non-increasing order, from most to least work. This ensures that the GPU cores are saturated with work by reducing the time that individual threads within a warp are idle, additionally reducing load imbalance between GPU cores at the end of the computation.
- As we solve the load balancing issue within individual kernels, we effectively improve the throughput of the self-join. As GPU's cores are active for more of the computation compared to previous work, we make better use of them.

This paper is organized as follows: Section II outlines background and related work, Section III presents the proposed solutions to mitigate load imbalance, Section IV evaluates the performance of our optimizations, and Section V concludes the work and discusses future work directions.

II. Background

In this section, we first formalize the problem of the distance similarity self-join, and then present related work. We also present the work we leverage and optimize.

A. Problem Statement

Let D be a dataset containing points in n dimensions. For every query point q_i, $i = 1, \dots, |D|$, we denote its coordinates as x_j, $j = 1, \dots, n$. Thus, $q_1(x_1)$ represents the coordinate in the first dimension of the point q_1. The distance similarity self-join will return all points $q_i \in D$ that are within the Euclidean distance ϵ of each other.

Let p, q be two points in D, where p is within ϵ of q if $dist(p, q) \leq \epsilon$, where $dist(p, q) = \sqrt{\sum_{a=1}^{n}(p(x_a) - q(x_a))^2}$. We elect to use the Euclidean distance as it is a common metric in low dimensionality and to facilitate a direct performance comparison with other implementations of the similarity self-join. All our processing occurs in-memory, and to avoid exceeding the GPU's global memory capacity, we use a batching scheme to incrementally compute the self-join result set across several batches. Moreover, we define a range query as the computation of the ϵ-neighborhood of a query point.

B. Related Work

Many studies have presented improvements to self-join performance. A common property between all these works is the use of the search-and-refine strategy to improve performance. The search part leverages a data indexing to bound the search space to candidate points that may be within ϵ of a query point. The refine part consists of computing the distance between the query point and its candidate points to only consider those within ϵ. We present an overview of related work regarding indexing, other similarity joins, and range queries.

1) Data Indexing: Based on the distance threshold, indexing allows for retrieving only those points that are likely to be within ϵ of a query point. Index efficacy is based on data properties. Therefore, an index designed for low dimensional data is unlikely to be suited for high dimensional data, and vice versa. Note that all index structures' efficacy degrades in higher dimensionality, which is why we focus on the low-dimensionality similarity self-join. Moreover, some of the indexing methods are suited to the CPU and are not directly applicable to the GPU without a significant performance loss due to several factors. As stated in Section I, hierarchical indexes such as trees and non-hierarchical structures such as grids are the two most prominent solutions for efficient indexing and pruning of the data, thus improving performance. We detail these two methods as follows.

Tree-based Indexing: Index-trees are widely used data structures for the similarity self-join and are particularly suited to those running on a CPU. These indexes are typically constructed based on the data distribution. The R-Tree [9] uses bounding boxes to partition data that are stored in the leaves of the tree. However, the use of bounding boxes as in [9] makes it not well suited to higher dimensions, as data are likely to be assigned to more than a single disjoint partition. Thus, some of the inner bounding boxes will have duplicate data due to their overlapping, leading to both an increase in memory usage and traversal time, as the number of paths traversed increases. Thus, the R*-Tree as proposed in [10] optimizes the area, the margin and the overlapping of these bounding boxes, while the X-Tree [11] improves this overlapping in higher dimensions. The k-d tree [8] is a binary tree organizing points in a k-dimensional space whose nodes are one of the two partitions of the space stored in their parent node. The k-d tree performs reasonably well in higher dimensions as there is no data duplication, each point is in a single disjoint partition.

Although the use of trees is not particularly suited to an efficient use on the GPU due to their many branch instructions and the recursive calls required to traverse them, several works address these issues to improve their efficiency on the GPU. Authors from [13] convert the recursive calls of the R-Tree into sequential accesses, while in [12] they optimize the R-Tree to execute on the GPU by also avoiding recursive calls, as well as improving the irregular memory accesses. In [14], a hybrid R-Tree using both the CPU and the GPU is proposed where the tree is traversed on the CPU, which then sends the data contained in the leaves to the GPU. The CPU performs the tree traversal, which has an irregular execution pattern, and the GPU performs the filtering of points in the leaf nodes. Hence, this exploits each architecture's relative strengths.

Grid-based Indexing: Statically partitioned grid-based indexing consists of partitioning the data into a grid of cells with length ϵ in each dimension. This data structure allows constraining the search of an ϵ-neighborhood of a query point to only its surrounding cells as proposed for a CPU implementation in [15]. Hence, in n dimensions, each point needs to consider up to 3^n cells. SUPER-EGO, as advanced in [16], is considered a state-of-the-art CPU parallel algorithm and uses a non-materialized grid indexing. In [18], the authors propose a grid indexing targeting the GPU, that only indexes non-empty cells. By doing so, the data structure memory footprint remains very low with a $O(|D|)$ space complexity. We thus compare our optimizations to SUPER-EGO [16] and this GPU solution [18].

2) Range Queries and Similarity Joins: The Locality-Sensitivity Hashing (LSH) algorithms, such as the E^{2}LSH algorithm [26]–[28], provides an estimated result of the nearest-neighbor search and can be used as an estimated distance similarity search [28] method working well in very high dimensions. However, we do not consider E^{2}LSH in this paper as it targets high-dimensional data and computes an estimated result, whereas we target lower dimensions and an exact result. The LSS algorithm as proposed in [24] also computes an estimation of the similarity-join by leveraging the use of a GPU and by using space-filling curves, turning the similarity join problem into a sort-and-search problem, which are two very efficient operations on the GPU. This technique creates the curves by sorting on the GPU; then each query point performs an interval search to find candidate points, efficiently pruning the search.

C. Overview of Leveraged Previous Work

We address load imbalance in the GPU self-join work of Gowanlock & Karsin [18]. We give a brief overview of their work, but refer the reader to additional details in [18]. In contrast to previous work, we focus on several kernel optimizations to mitigate load imbalance.

1) GPU Grid Index: We reuse the ϵ grid indexing for the GPU as proposed in [18]. This method uses several arrays to efficiently store the data into cells of length ϵ. When performing a range query around a query point, this technique bounds the search to only adjacent grid cells. Moreover, as their method only indexes the non-empty cells, the memory footprint is very low, having a space complexity of $O(|D|)$, making it well-suited to the GPU's limited memory capacity. Moreover, each thread performs the same bounded search by accessing neighboring cells, thus reducing the divergence of the threads within the same warp. We reuse their index, which we denote as I.

2) Batching Scheme: Depending on the dataset and the value of ϵ, the self-join may generate a result set size exceeding the GPU's global memory capacity. In [17], [18], the authors advance a solution to prevent buffer overflow. Through a sequence of batches consisting of multiple kernel executions, they compute the self-join while not exceeding the GPU's global memory capacity by transferring partial results back to the host. Thus, with a combination of multiple kernel invocations, pinned memory, and GPU streams, they avoid all global memory buffer overflow and are also able to hide data transfer overhead by overlapping them with kernel executions. This technique samples a percentage of the dataset to estimate the total result set size, yielding the number of batches that need to be executed. In this work, we sample 1% of the entire dataset. The number of batches, $nbBatches$, is determined by the desired maximum result set size for each kernel execution of size b_s. In this paper, we fix this value $b_s = 10^8$ and we use 3 streams. Thus, when using 3 streams, the total pinned memory buffer size is 3×10^8. We define a *batch* as one kernel invocation of the self-join which returns a partial result set, where several batches are needed to compute the entire self-join result. Moreover, we define D_l as the data points assigned to the *batch* l, where $l = 0, 1, \dots, nbBatches - 1$.

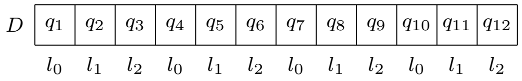


Fig. 1. Example representation of the thread assignment across multiple *batches*. $|D| = 12$, split across 3 *batches* $numbatch = 3$ with 4 points each. l corresponds to the *batch* computing the query point $q_i \in D$, where $i = 1, \dots, 12$ and $l = (i-1) \; mod \; nbBatch$.

Figure 1 represents an example of how threads are assigned across multiple *batches*. We use for this example a dataset D of 12 points and 3 *batches*, $nbBatches = 3$. Therefore, each *batch* has 4 points and thus 4 threads, which are strided across the dataset. Hence, the query point $q_i \in D$ is computed by the *batch* $l = (i-1) \; mod \; nbBatches$.

3) GPU Kernel: The GPUCALCGLOBAL kernel, as advanced by Gowanlock & Karsin in [18], is the foundation of several of our optimizations. This kernel computes the ϵ-neighborhood of each point in a dataset D, and where each query point in D is computed by a single thread on the GPU. Thus, $|D|$ threads are used. This kernel is given in Algorithm 1, and is a slightly modified version from [18] to use an index I instead of defining each index component described in [18]. The kernel first retrieves the thread's global id (determined by the block's id, the block's size and the thread id within its block), then returns if the thread's global id is larger than the size of the *batch*, as it uses one thread per query point (lines 2 and 3). On line 5, the thread gets its point corresponding to its global id, as well as the neighboring cells on line 6. Then, for each neighboring cell that was found (line 7), we retrieve the list of points contained in the cell (line 8). Afterward, for each candidate point from the neighboring cell (line 9), we compute the distance between this candidate point and the query point (line 10). If the candidate point is within ϵ (line 11), then we add the pair of both points to the result set (line 12). Finally, when a query point has completed its distance calculations, the kernel returns (line 13).

*4) Unidirectional Comparison (*UNICOMP*):* Gowanlock & Karsin [18] have advanced a cell access pattern designed to eliminate any duplicate calculations between the points for datasets in any dimension. As the Euclidean distance is a symmetric function ($dist(p, q) = dist(q, p)$), they can add both pairs of points to the result set, with only one distance calculation. This cell access pattern, although presenting improved response time in most of their experimental evaluations, seems to present an uneven workload balance

Algorithm 1 GPUCalcGlobal Kernel (GPU) from [18]

```
procedure GPUCALCGLOBAL($D_l, I, \epsilon$)
    gid ← getGlobalId()
    if gid ≥ $|D_l|$ then return
    resultSet ← ∅
    point ← $D_l$[gid]
    adjCells ← getNeighboringCells(gid)
    for cell ∈ adjCells do
        pointsArr ← getPoints(cell)
        for candidatePoint ∈ pointsArr do
            result ← calcDistance(point, candidatePoint, $\epsilon$)
            if result ≠ ∅ then
                atomic: gpuResultSet ← gpuResultSet ∪ result
    return
```

between threads. For example, in two dimensions, a point may compare with points in up to eight adjacent cells, whereas some points may not compare to any adjacent cell. The implementation of this UNICOMP cell access pattern is given in two dimensions in Algorithm 2, as described in [18]. UNICOMP relies on the odd multidimensional coordinates of the cells to establish an access pattern. It takes as input the query point $point$, C_a its multidimensional coordinates, $filteredRngs$ the range of the non-empty cells in each dimension, and B the array of non-empty cells. If the first coordinate of a cell is odd (line 2), then this cell is a part of the pattern. The algorithm iterates over the first dimension (line 3), and if the explored cell does not have the same first index as the origin cell (line 4), then the linear id of the neighboring cell is calculated (line 5). If this linear id corresponds to a non-empty cell (line 6), the origin point is compared to the points of the neighboring cell (line 7). Lines 8 to 14 are used to iterate over the second dimension. As this example is for two dimensions indexing, an additional loop is needed for each additional dimension. The comments "Green arrows" and "Red arrows" (respectively lines 2 and 8) refer to the arrows represented in Figure 2.

Algorithm 2 The Unicomp cell access pattern in 2 dimensions (GPU) from [18]

```
 1: procedure UNICOMP2D($point, C_a, filteredRngs, B$)
 2:     if $C_a$.x is odd then                              ▷ Green arrows
 3:         for x ∈ $filteredRngs[1]$ do
 4:             if x ≠ $C_a$.x then
 5:                 linearID ← getLinearCoord(x, $C_a$.y)
 6:                 if linearID ∈ $B$ then
 7:                     ComparePoints($point$, linearID)
 8:     if $C_a$.y is odd then                              ▷ Red arrows
 9:         for x ∈ $filteredRngs[1]$ do
10:             for y ∈ $filteredRngs[2]$ do
11:                 if y ≠ $C_a$.y then
12:                     linearID ← getLinearCoord(x, y)
13:                     if linearID ∈ $B$ then
14:                         ComparePoints($point$, linearID)
15:     return
```

Figure 2 represents the cell access pattern of Unicomp in two dimensions. The arrows represent the neighboring cells to compare to, while the numbers in the cells quantify the number of neighboring cells that are compared.

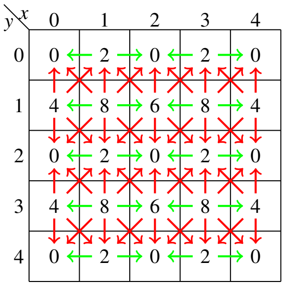


Fig. 2. UNICOMP cell access pattern in two dimensions. The numbers represent the number of neighboring cells the origin cell is going to compare, and the arrows indicate these neighboring cells. While green arrows indicate an odd x index, red arrows are for an odd y index.

III. Mitigating Load Imbalance

The SIMT architecture of the GPU makes it well-suited for highly data-parallel applications. Threads are executed in groups of 32 called warps [25]. Due to hardware limitations (e.g., the number of available registers), only a limited number of warps can be executed concurrently on the GPU. In the case of the distance similarity self-join, the workload is dependent on the data distribution, therefore potentially disparate within threads of the same warp. For example, in a real world dataset, some points will have few neighbors, and some will have many neighbors, potentially spanning several orders of magnitude. In this situation where threads of the same warp have both points from a dense region and points from a sparse region, some of these threads will be idle for a longer amount of time than others. While the threads computing the points from a dense region of the dataset are still active, this prevents the execution of a new warp.

Figure 3 is an example representation of the possible workload imbalance we might face within a warp when using the original GPUCALCGLOBAL kernel described in Section II-C3. Due to intra-warp workload imbalance, some of the threads will be idle while some others will be computing, thus reducing the GPU's resources usage efficiency.

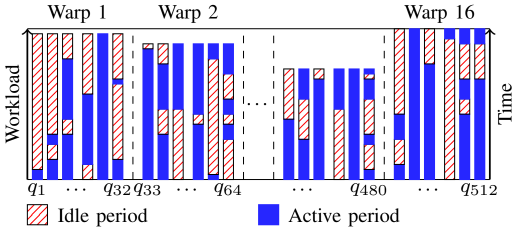


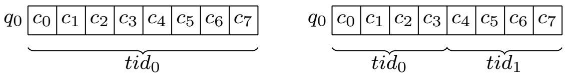
Fig. 3. Example representation of the workload across a dataset, with q_1 to q_{32} query points in the first warp, q_{33} to q_{64} query points in the second warp, and q_{481} to q_{512} in the last warp, assuming $|D| = 512$. This represents the potential workload imbalance of the original GPUCALCGLOBAL kernel.

A. Increasing the Granularity of each Range Query

The GPU kernel advanced by Gowanlock & Karsin [18] uses a single GPU thread per query point. Thus, a single thread is computing every distance calculation between its point and all the neighboring points. Depending on the properties of the data, some query points may have many distance calculations to compute, and therefore large amounts of work. Consequently, if one thread is assigned to compute all of the distance calculations, then at the end of a kernel execution, some of the GPU cores will be idle. Therefore, we can increase the granularity of the filtering task by assigning multiple threads to each query point for computing the distances. This reduces the amount of idle resources at the end of the computation.

An optimization is to use multiple threads per query point, so each thread is computing a fraction of the distance calculations of its assigned query point. This will reduce the workload of each thread, and thus reduce the time needed to find the neighbors of the query point. Moreover, by assigning the same workload to each thread of a query point, and as the number of thread within a warp is fixed, using this optimization will reduce the intra-warp load imbalance. However, increasing the total number of threads implies a larger number of warps to schedule. We define k as the number of threads assigned to a single query point.

Figure 4 represents how we assign threads to the candidate points c_i of a query point q_j. In the present case, we use as an example a query point q_0 with a neighboring cell containing eight points: c_0 to c_7, and $k = 2$. Figure 4 (a) represents how all candidate points are assigned to the single thread as in the GPUCalcGlobal kernel, while Figure 4 (b) represents the candidate points being assigned to the two different threads. For descriptive purpose, we assume k evenly divides the number of candidate points.

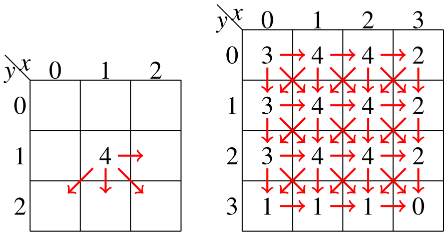


(a) (b)

Fig. 4. (a) Original assignment of a thread to candidate points, as in [18], (b) Assignment of threads to candidate points when increasing the distance calculation granularity, with $k = 2$ (even case). q_0 is a query point with eight candidate points ($\{c_0,\ldots,c_7\}$). tid_i designates the local thread id of the query point, where $i = 0, \ldots, k - 1$.

B. Cell Access Pattern: Linear ID Unidirectional Comparison

We propose Linear ID Unidirectional Comparison (Lid-Unicomp) as an optimization to the Unicomp cell access pattern advanced in [18]. While Unicomp relies on extensive conditional statements to determine whether points of a cell need to compare to neighboring cells, Lid-Unicomp reuses the fact that with the grid indexing we use, non-empty cells have a unique linear id computed from the cell's coordinates in n dimensions. The principle of this new cell access pattern is thus to compute the distance with the points from every neighboring cell that has a higher linear id than the origin cell. The cell access pattern of this method is represented in Figure 5. As we observe if we compare Figure 2 to Figure 5, when using the Lid-Unicomp pattern, every inner cell will compare to the same number of neighboring cells. Depending on the data distribution, this might greatly improve the workload balance over Unicomp because it has some cells comparing to every neighboring cell, and some cells comparing to none.



(a) (b)

Fig. 5. Overview of the Lid-Unicomp pattern in 2-D. The numbers represent the number of neighboring cells the origin cell is going to compare to, and the arrows indicate these neighboring cells. (a) represents the cell access pattern on its own in 2-D, (b) represents its application on a 2-D grid.

Algorithm 3 gives the implementation of the Lid-Unicomp cell access pattern. For each neighboring cell (line 3), if the linear id of the neighboring cell is greater than the linear id of the origin cell (line 5), then following the Lid-Unicomp cell access pattern, this cell needs to be evaluated (line 6).

In comparison with the Unicomp cell access pattern (Section II-C4), the implementation of the Lid-Unicomp pattern is more straightforward as it relies on a linear id calculation and a single condition, whereas Unicomp relies on an extensive combination of loops and conditions. An advantage of Unicomp is that it stops searching as soon as an iterated multidimensional coordinate is even (Algorithm 2, lines 2 and 8), while Lid-Unicomp checks the linear id of all the non-empty adjacent cells. Consequently, Unicomp does not need to iterate over all adjacent cells in comparison to Lid-Unicomp. Thus, Unicomp may outperform Lid-Unicomp, depending on several data-dependent factors.

Algorithm 3 Lid-Unicomp cell access pattern implementation (GPU)

```
1: procedure LidUnicomp($q, originCell, \epsilon$)
2:     originId ← linearId($originCell$)
3:     for c ∈ getNeighborCells($originCell$) do
4:         neighborId ← linearId(c)
5:         if originId < neighborId then
6:             evaluateNeighborCell($q$, c)
```

C. Local and Global Load Balancing: Sorting by Workload

Consider two threads t_0 and t_1, where t_0 is assigned a query point in a sparse region (q_0 in Figure 6), and t_1 is assigned a query point in a dense region (q_1 in Figure 6). t_0 will perform 14 distance calculations, and t_1 will perform 45 distance calculations. If these threads are within the same warp, t_1 will have much more work than t_0, and t_0 will be idle for a significant amount of time as it waits for t_1.

To reduce the amount of time that threads are idle, a solution is to sort the points by their workload (number of point comparisons), such that each warp be assigned threads with similar workloads in comparison to an unbalanced workload as in Figure 3. This sorting is achieved by computing the number of distance calculations of each non-empty cell, i.e., retrieving their number of neighbors, and assigning points from the cell

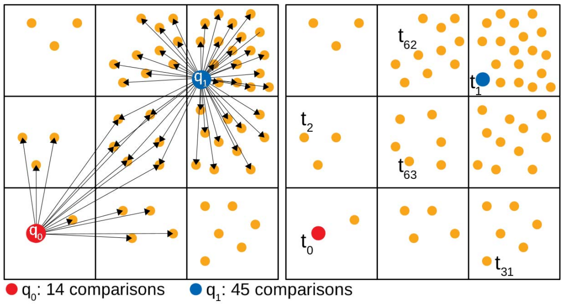


Fig. 6. Illustration of the load imbalance between query points and therefore between threads, where q_0 and q_1 are two query points and t_0 and t_1 two threads processing the query points q_0 and q_1 respectively.

with the greatest workload at the beginning of a new array denoted as D'. Furthermore, as a consequence of the batching scheme (Section II-C2), the data points assigned to each *batch* D_l have a similar total workload due to accessing the data in a strided manner (Figure 1). SORTBYWL is applied to each *batch* D'_l and not D'; therefore warps will not be strictly assigned points from most workload to least work in the scope of the entire dataset. However, this ensures that each *batch* does not overflow the result set buffer.

D. *Forcing the Warp Execution Order using a Work Queue*

SORTBYWL does not entirely obviate load imbalance as threads within the same warp still have different workloads due to the stride of the threads across the dataset as presented in Section II-C2. Moreover, the hardware scheduler may not execute the warps from most workload to least work, as the scheduler still has control over the execution order of warps. To obviate these issues, we propose using a priority queue. While previous work implementing a queue on the GPU exists [29], they use a distributed queue with dynamic load balancing where threads can retrieve or give work to other threads. Moreover, they use their threads for the entire computation duration, making it unsuitable for our work due to our batching scheme. Therefore, we do not use the queue from [29] and extend our SORTBYWL optimization using a queue that is persistent across all kernel invocations. Thus, complementary to SORTBYWL which outputs a sorted array of the points, we use a global counter to indicate the equivalent of the head of a queue. Each thread increments this counter through an atomic operation that assigns data points to threads. By using this optimization, we expect our workload to be nearly identical between the threads of the same warp, as the example represented in Figure 7, where the idle periods of the threads are significantly reduced in comparison to Figure 3.

Figure 8 represents the functioning of our WORKQUEUE optimization, where D' is our dataset sorted by workload and W indicates the total workload of a query point. The workload is quantified as the number of distance calculations a query point will perform to refine its candidate set.

Unlike SORTBYWL, we consider the entire dataset D' (as sorted by workload) when executing *batches*, and do not employ adding points to D_l across *batches* in a strided manner.

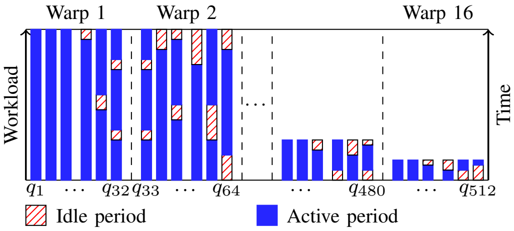
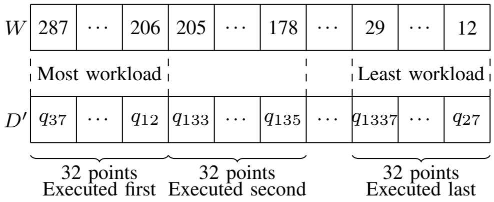

Warp 1
Warp 2
Warp 16
Workload
Time
q_1 ··· q_{32} q_{33} ··· q_{64} ··· q_{480} ··· q_{512}
▨ Idle period
■ Active period

Fig. 7. Representation of balancing the workload between the threads within the same warp. We use for this example a dataset $D = 512$.



W	287	···	206	205	···	178	···	29	···	12
	Most workload							Least workload		
D'	q_{37}	···	q_{12}	q_{133}	···	q_{135}	···	q_{1337}	···	q_{27}
	32 points Executed first			32 points Executed second				32 points Executed last		

Fig. 8. Representation of the points' execution order when using the WORKQUEUE optimization. D' is the sorted dataset, and W gives the workload of each query point. The first 32 points with the most workload will be executed at the beginning within the same warp, while the last 32 points, with the least workload, will be executed at the very end.

This ensures that each warp has the smallest possible variance in workloads (point comparisons). However, this leads to large variance in result set sizes across *batches*, which the strided D_l *batches* were designed to avoid. Consequently, in the WORKQUEUE optimization, we slightly modify the batching scheme (Section II-C2), and instead of sampling the entire dataset to estimate the total result set size, we sample the first 1% of D' (without striding), which yields a much larger estimated total result set size. This ensures that our first *batch* does not overflow the result set buffer; however, we execute more batches than when using GPUCALCGLOBAL or SORTBYWL.

Finally, when we use the WORKQUEUE in combination with a $k > 1$, we use cooperative groups introduced with CUDA 9.0 [30]. We thus create groups of size k where only the first thread increments the global counter and then shuffles the returned result to the other threads of the cooperative group.

IV. EXPERIMENTAL EVALUATION

A. *Datasets*

To evaluate our proposed solutions, we select several datasets presenting different characteristics such as the dimensionality and size. We consider datasets synthetically generated with a uniform, and an exponential distribution with $\lambda = 40$, each composed of two million points between two and six dimensions. We use these both distributions as they present opposite workloads, and therefore to outline the impact of our optimizations. For the real world datasets, we use the *SW-* datasets [31] with 1.86M and 5.16M points, both in two and three dimensions representing the latitude and longitude of the objects in two dimensions, including the total number of electrons in the ionosphere as the third dimension. Moreover, we select 50 million points from the *Gaia* catalog [32] in two dimensions. For the synthetic datasets, we denote *Expo-* as exponentially distributed datasets and *Unif-* as uniformly distributed datasets. The summary of these datasets is given in Table I. We omit the three, four and five-dimensional data of our synthetic datasets for the intermediate plots (Figures 9, 10 and 11), as two and six dimensions bound the performance.

TABLE I
Summary of the different datasets used for the experimental evaluation. $|D|$ denotes the number of points and n the dimensionality.

Dataset	$\|D\|$	n	Dataset	$\|D\|$	n	Dataset	$\|D\|$	n
Unif2D2M	2M	2	*Expo2D2M*	2M	2	*SW2DA*	1.86M	2
Unif3D2M	2M	3	*Expo3D2M*	2M	3	*SW2DB*	5.16M	2
Unif4D2M	2M	4	*Expo4D2M*	2M	4	*SW3DA*	1.86M	3
Unif5D2M	2M	5	*Expo5D2M*	2M	5	*SW3DB*	5.16M	3
Unif6D2M	2M	6	*Expo6D2M*	2M	6	*Gaia*	50M	2

B. Methodology

We use a platform composed of 2×Intel E5-2620v4@2.10 GHz for a total of 16 cores, coupled with 128 GiB of RAM and an Nvidia Quadro P100 (16 GiB of HBM2 global memory). The GPU code is written in CUDA, while the C/C++ host code is compiled with the GNU compiler with the O3 flag. The response times do not include the index construction time because we do not optimize index construction in the implementations that we compare to. All other components of the algorithm are included in the response time.

In all GPU experiments, we use 256 threads per block, and each data point is represented as a 64-bit floating point. The parallel CPU Super-EGO experiments include the time to EGO-sort and join, and use 32-bit floating points and run using 16 threads across 16 physical cores, yielding the best configuration on our platform. Table II outlines the optimizations and notation used in the experimental evaluation.

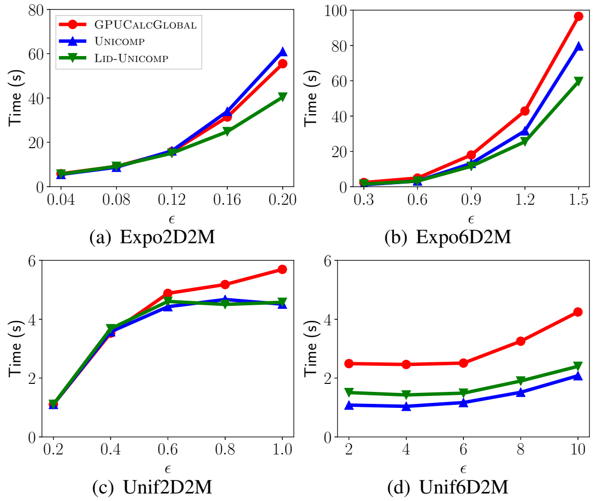
TABLE II
Optimizations and notation used throughout the evaluation.

Notation	Description
GPUCalcGlobal	Original GPU kernel [18] we compare to.
Unicomp	Original cell access pattern [18] we compare to.
Super-EGO	State-of-the-art CPU parallel algorithm [16] that we compare to.
Lid-Unicomp	Cell access pattern advanced in Section III-B.
SortByWL	Sorting by workload optimization (Section III-C).
WorkQueue	Work-queue optimization (Section III-D).
k	Number of thread per query point (Section III-A).

We average the response times over three trials, while we profile on only three *batches* as each *batch* has nearly identical performance characteristics. Although we retrieve several different metrics through the Nvidia Profiler [33], we choose only to report the warp execution efficiency in this paper, as it is the most relevant metric among those we have collected regarding the performance of our optimizations.

C. Results

Impact of the New Cell Access Pattern: Here, we evaluate the response time of the Lid-Unicomp cell access pattern optimization (Section III-B) in several scenarios and compare it to the response time of the GPUCalcGlobal and Unicomp kernels, the two solutions we aim to improve. Figure 9 plots the response time vs. ϵ of the GPUCalcGlobal kernel, and the Unicomp and Lid-Unicomp cell access patterns on our uniformly and exponentially distributed datasets, in two and six dimensions. We observe that Unicomp has a lower response time than GPUCalcGlobal, excepting the *Expo2D2M* dataset when $\epsilon > 0.12$ (Figure 9 (a)). Moreover, our solution, Lid-Unicomp, improves the performance of the self-join in most cases, except on the *Unif6D2M* dataset (Figure 9 (d)).



(a) Expo2D2M (b) Expo6D2M (c) Unif2D2M (d) Unif6D2M

Fig. 9. Response times of the Lid-Unicomp cell access pattern, versus the GPUCalcGlobal kernel and the Unicomp cell access pattern on our synthetic datasets. The legend in (a) is used across all subfigures, and we set $k = 1$.

To understand these results, we profile the execution of these three configurations, on the *Expo2D2M*, *Expo6D2M*, *Unif2D2M*, and *Unif6D2M* with $\epsilon = 0.2, 1.2, 1.0, 8.0$, respectively. We report the results in Table III. The warp execution efficiency is the average percentage of active threads in each executed warp. We choose this metric as having a high warp execution efficiency means that only a few threads are idle during the execution of each warp. The warp execution efficiency between Unicomp and Lid-Unicomp is correlated to the response time. In most cases, as the warp execution efficiency is higher for Lid-Unicomp than Unicomp, the response time is lower, with an exception for the *Unif6D2M* dataset (Figure 9 (d)). Regarding the GPUCalcGlobal kernel, despite a higher warp execution efficiency than the Unicomp or Lid-Unicomp optimizations, its response time is higher. This is because both cell access patterns reduce the number of distance calculations by a factor of roughly two, thus improving the response time. Thus, the proposed Lid-Unicomp optimization may be more efficient than the previous UNICOMP cell access pattern due to its more evenly distributed work across threads. Moreover, the low warp execution efficiency on the exponentially distributed datasets may reflect intra-warp workload imbalance.

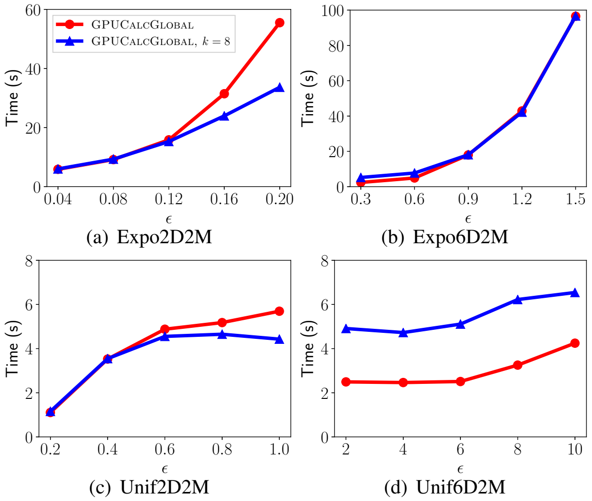
TABLE III
WARP EXECUTION EFFICIENCY (WEE) OF THE GPUCALCGLOBAL KERNEL AS WELL AS THE UNICOMP AND LID-UNICOMP CELL ACCESS PATTERNS OVER OUR SYNTHETIC DATASETS AND FOR SPECIFIC VALUES OF ϵ. THE TIME CORRESPONDS TO THAT IN FIGURE 9.

		GPUCalcGlobal		**Unicomp**		**Lid-Unicomp**	
Dataset	ϵ	WEE(%)	Time(s)	WEE(%)	Time(s)	WEE(%)	Time(s)
Expo2D2M	0.2	26.5	55.5	13.2	60.9	18.3	40.4
Expo6D2M	1.2	15.2	42.9	7.8	31.6	10.0	25.5
Unif2D2M	1.0	75.4	5.7	48.94	4.5	69.1	4.6
Unif6D2M	8.0	51.3	3.3	19.25	2.1	40.9	2.4

Impact of Assigning Multiple Threads to Each Query Point: We now focus on the performance of increasing the thread granularity, specifically by using eight threads per point ($k = 8$). We compare this optimization to the GPUCALCGLOBAL kernel, which uses only one thread ($k = 1$), and use the same datasets as for the LID-UNICOMP performance evaluation. Having $k > 1$ reduces the workload of each thread, by reducing the number of distance calculations each of them has to compute. Moreover, this also reduces the workload variance within a warp, as the threads computing the same query point will share the same total workload. Figure 10 plots the response time of the GPUCALCGLOBAL kernel when $k = 1$ and when $k = 8$ on our synthetic datasets. While the *Expo2D2M* dataset (Figure 10 (a)) greatly benefits from the increased granularity when $\epsilon \geq 0.12$, the response time is not impacted on the *Expo6D2M* dataset (Figure 10 (b)) and performs even worse when $\epsilon \leq 0.9$. Regarding the uniformly distributed datasets, while *Unif2D2M* presents a lower response time when having $k = 8$ and $\epsilon \geq 0.4$ (Figure 10 (c)), the GPUCALCGLOBAL kernel with $k = 1$ performs better on the *Unif2D2M* dataset (Figure 10 (d)). Therefore, having a low workload as it is the case for lower values of ϵ seems not to be suited to an increase of the workload granularity, although it does not especially degrade performance. The exception is on the *Unif6D2M* dataset, which performs better when $k = 1$ for every ϵ values.

Table IV shows the warp execution efficiency and the response time of the GPUCALCGLOBAL kernel when $k = 1$ and $k = 8$. We observe that having more threads greatly increases the warp execution efficiency, particularly for the exponentially distributed datasets. This observation is reflected in the response time, which is lower for our selected values of ϵ. However, although the warp execution efficiency is always higher when $k = 8$, the response time of this configuration is higher on the *Unif6D2M* dataset than for $k = 1$. We leave investigating this behavior for future work.

Impact of Reordering the Points by Workload and Forcing Warp Execution Order: We evaluate the performance of our SORTBYWL and WORKQUEUE optimizations, compared to GPUCALCGLOBAL. Figure 11 plots the response time vs. ϵ of the GPUCALCGLOBAL kernel, and our SORTBYWL and WORKQUEUE optimizations on our uniformly and exponentially distributed datasets, in two and six dimensions. Observing the exponentially distributed datasets in two and six dimensions (Figures 11 (a)-(b)), we see an improvement in the response time, particularly for higher values of ϵ. For smaller values of ϵ, the workload variance between points is reduced, thus decreasing the impact of packing the points based on their workload. Moreover, even without controlling the execution workflow when using the SORTBYWL optimization, it performs better than GPUCALCGLOBAL in every case on the exponentially distributed datasets. Nevertheless, the WORKQUEUE thus seems to be very effective, especially on datasets with significant variance of workload between points, as expected. However, sorting the points based on their workload does not present any significant gain when datasets are uniformly distributed as every point have a similar workload, unlike exponentially distributed datasets. We observe this on Figures 11 (c)-(d)) where neither SORTBYWL or WORKQUEUE significantly outperform GPUCALCGLOBAL.



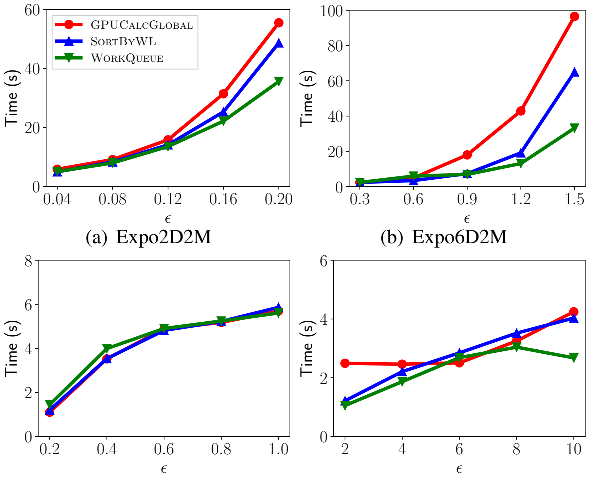
(a) Expo2D2M
(b) Expo6D2M
(c) Unif2D2M
(d) Unif6D2M

Fig. 10. Response time of the increase of the granularity when $k = 8$ versus $k = 1$ for the GPUCALCGLOBAL kernel on our synthetic datasets. The legend in (a) is used across all subfigures.

TABLE IV
WARP EXECUTION EFFICIENCY (WEE) OF THE GPUCALCGLOBAL WITH $k = 1$ AND WHEN $k = 8$ ON SYNTHETIC DATASETS AND FOR SPECIFIC VALUES OF ϵ. THE TIME CORRESPONDS TO THAT IN FIGURE 10.

		GPUCalcGlobal		**GPUCalcGlobal, $k = 8$**	
Dataset	ϵ	WEE (%)	Time (s)	WEE (%)	Time (s)
Expo2D2M	0.2	26.5	55.5	40.8	33.6
Expo6D2M	1.2	15.2	42.9	39.27	42.2
Unif2D2M	1.0	75.4	5.7	80.3	4.4
Unif6D2M	8.0	51.3	3.3	60.9	6.2

In Table V, we observe that the warp execution efficiency is much higher when using WORKQUEUE. Moreover, an increase of the warp execution efficiency results in a decrease of the response time, excepting the *Unif2D2M* dataset. The WORKQUEUE presents both the highest warp execution efficiency and the lowest response time on our selected configu-



(a) Expo2D2M (b) Expo6D2M

(c) Unif2D2M (d) Unif6D2M

Fig. 11. Response time of the SortByWL and WorkQueue optimizations against the GPUCalcGlobal kernel on our synthetic datasets. The legend in (a) is used across all subfigures, and we set $k = 1$.

rations. Therefore, our strategy of packing warps with similar workloads and forcing the hardware scheduler to execute warps in order clearly improves performance.

TABLE V
Warp execution efficiency (WEE) of the GPUCalcGlobal kernel, as well as the SortByWL and WorkQueue optimizations on our synthetic datasets and for specific values of ϵ. The time corresponds to that in Figure 11.

		GPUCalcGlobal		**SortByWL**		**WorkQueue**	
Dataset	ϵ	WEE(%)	Time(s)	WEE(%)	Time(s)	WEE(%)	Time(s)
Expo2D2M	0.2	26.5	55.5	74.6	48.7	83.2	35.6
Expo6D2M	1.2	15.2	42.9	71.4	19.1	95.6	13.1
Unif2D2M	1.0	75.4	5.7	75.4	5.9	83.1	5.6
Unif6D2M	8.0	51.3	3.3	48.2	3.5	48.4	3.0

Combination of Approaches on Real World Datasets: Figure 12 plots the response time vs. ϵ using a combination of our optimizations, including WorkQueue with Lid-Unicomp and $k = 8$. This combination of optimizations outperforms GPUCalcGlobal and Super-EGO across nearly all experimental scenarios. In particular, our optimizations are the most effective on the largest workloads (large datasets and ϵ). The performance on the real world datasets is limited to $n = 3$ dimensions. Thus, the performance of our optimizations typically converge across the datasets because the workloads are low at $n \leq 3$ dimensions, but we will show that on higher dimensionality (Figure 13), the combination of all optimizations will yield larger performance gains (e.g., the *Expo6D2M* dataset, Figure 11 (b)).

Table VI shows the warp execution efficiency and the total response time for selected values of ϵ from Figure 12. All of our solutions present a better warp execution efficiency and overall response time than GPUCalcGlobal, which indicates that warp execution efficiency is a good metric for assessing load imbalance. Due to the high warp execution efficiency observed across all of our optimizations in Table VI, we believe that further optimizations to the GPUCalcGlobal kernel are not likely to lead to significant performance gains. However, new algorithmic designs may improve performance.

TABLE VI
Warp execution efficiency (WEE) of the GPUCalcGlobal, the WorkQueue, and the WorkQueue combined with Lid-Unicomp and $k = 8$ on our real world datasets and for specific values of ϵ. The time corresponds to that in Figure 12.

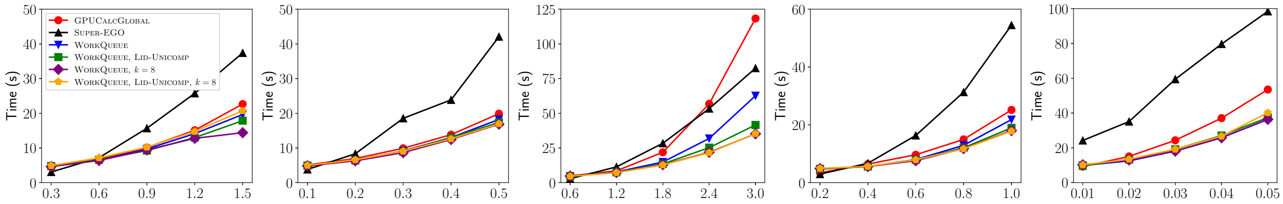
		GPUCalcGlobal		**WorkQueue, Lid-Unicomp**		**WorkQueue, $k = 8$ Lid-Unicomp**	
Dataset	ϵ	WEE(%)	Time(s)	WEE(%)	Time(s)	WEE(%)	Time(s)
SW2DA	1.2	55.2	15.1	89.1	13.0	80.7	12.5
SW2DB	0.4	54.2	13.8	83.0	13.0	79.7	12.6
SW3DA	2.4	33.7	56.8	93.4	25.2	83.2	21.6
SW3DB	0.8	40.8	14.9	87.1	12.1	82.5	11.7
Gaia	0.04	64.1	37.1	80.3	27.1	78.3	26.7

V. Discussion and Conclusion

The self-join has data-dependent performance behavior and irregular instructions that make the problem challenging to solve efficiently on the GPU. Depending on the data distribution, the self-join leads to load imbalance within each warp, which limits the GPU's throughput. Consequently, we have advanced several optimizations that address load imbalance. We propose a cell access pattern that avoids duplicate computation. In contrast to previous work, this allows each point in the dataset to be compared to the same number of adjacent cells. We increase the granularity of each range query by assigning each query point multiple threads for performing the distance calculations. This reduces the number of varying workloads within each warp. We propose packing warps with threads assigned similar workloads to reduce the load imbalance within each warp. Lastly, we ensure that the GPU's hardware scheduler executes warps in non-increasing order of each warp's assigned work. This reduces inter-warp load imbalance, which ensures that the warps finish their execution at similar times, at the end of the kernel execution.

Figure 13 summarizes the performance of our WorkQueue, Lid-Unicomp and $k = 8$ optimizations combined on all datasets. From the figure, we find that using the optimizations outlined in this paper, we are able to significantly improve the performance over (a) a parallel CPU implementation, and (b) previous GPU self-join work. We achieve speedups up to 10.7× over Super-EGO and 9.7× over GPUCalcGlobal, with an overage of 2.5× and 1.6× respectively. This work demonstrates that reducing intra-warp workload imbalance can significantly improve performance, and thus has implications for other algorithms with data-dependent performance characteristics.

Future work directions are outlined as follows. We will apply our optimizations to other applications, especially the WorkQueue, which could be adapted to any self-join indexing structure, contingent upon being able to quantify the workload. We will investigate dynamically grouping *batches* of queries together when using the work queue such that each *batch* yields similar result set sizes. Additionally, we will carry



(a) SW2DA (b) SW2DB (c) SW3DA (d) SW3DB (e) Gaia

Fig. 12. Response time vs. ϵ on real world datasets of the WorkQueue optimization, the WorkQueue combined with the Lid-Unicomp pattern, the WorkQueue with $k = 8$, and both combined to the WorkQueue compared to the GPUCalcGlobal kernel and the Super-EGO CPU parallel algorithm. The legend in the subfigure (a) is used across all subfigures.

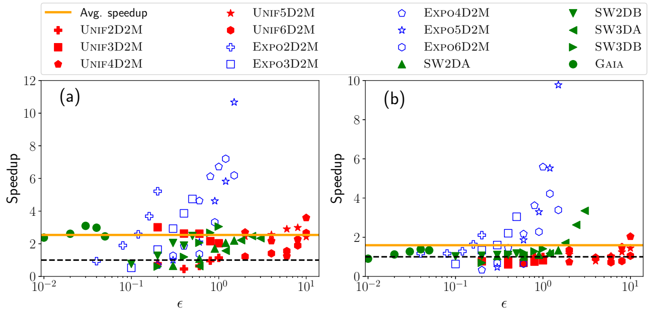


Fig. 13. Speedup of the WorkQueue combined to Lid-Unicomp and $k = 8$ optimization against the Super-EGO parallel algorithm (a), and over the GPUCalcGlobal kernel (b), on several datasets. ϵ values are plotted on a log scale to observe all data points.

out a more extensive performance comparison between the proposed cell access pattern and the one proposed by our previous work.

References

[1] S. Chaudhuri, V. Ganti, and R. Kaushik, "A Primitive Operator for Similarity Joins in Data Cleaning," *22nd Intl. Conf. on Data Engineering*, 2006.

[2] C. Xiao, W. Wang, X. Lin, J. X. Yu, and G. Wang, "Efficient Similarity Joins for Near-duplicate Detection," *ACM Trans. Database Syst.*, vol. 36, no. 3, pp. 15:1–15:41, 2011.

[3] R. Baraglia, G. D. F. Morales, and C. Lucchese, "Document Similarity Self-Join with MapReduce," pp. 731–736, 2010.

[4] C. Böhm, B. Braunmller, M. M. Breunig, and H.-P. Kriegel, "High Performance Clustering Based on the Similarity Join." *Proc. of the Intl. Conf. on Information and Knowledge Management*, pp. 298–305, 2000.

[5] C. Böhm and F. Krebs, "The k-Nearest Neighbour Join: Turbo Charging the KDD Process," *Knowledge and Information Systems*, vol. 6, no. 6, pp. 728–749, 2004.

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

[7] R. Elmasri and S. B. Navathe, *Fundamentals of Database Systems*, 7th ed. Pearson, 2016, ch. 18.4.1.

[8] J. L. Bentley, "Multidimensional Binary Search Trees Used for Associative Searching," *Commun. ACM*, vol. 18, no. 9, pp. 509–517, 1975.

[9] A. Guttman, "R-trees: A Dynamic Index Structure for Spatial Searching," *SIGMOD Rec.*, vol. 14, no. 2, pp. 47–57, 1984.

[10] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger, "The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles," vol. 19, 1990.

[11] S. Berchtold, D. A. Keim, and H.-P. Kriegel, "The X-tree : An Index Structure for High-Dimensional Data," 1996.

[12] J. Kim, W. Jeong, and B. Nam, "Exploiting Massive Parallelism for Indexing Multi-Dimensional Datasets on the GPU," *IEEE Trans. on Parallel and Distributed Systems*, vol. 26, no. 8, pp. 2258–2271, 2015.

[13] J. Kim, S.-G. Kim, and B. Nam, "Parallel multi-dimensional range query processing with R-trees on GPU," *Journal of Parallel and Distributed Computing*, vol. 73, no. 8, pp. 1195 – 1207, 2013.

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

[15] C. Böhm, B. Braunmüller, F. Krebs, and H.-P. Kriegel, "Epsilon Grid Order: An Algorithm for the Similarity Join on Massive High-dimensional Data," *SIGMOD Rec.*, vol. 30, no. 2, pp. 379–388, 2001.

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

[17] M. Gowanlock, C. M. Rude, D. Blair, J. D. Li, and V. Pankratius, "Clustering Throughput Optimization on the GPU," *Proc. of the 31st IEEE Intl. Parallel & Distributed Processing Symposium*, pp. 832–841, 2017.

[18] M. Gowanlock and B. Karsin, "GPU Accelerated Self-join for the Distance Similarity Metric," *Proc. of the 2018 IEEE Intl. Parallel and Distributed Processing Symposium Workshops*, pp. 477–486, 2018.

[19] R. Bellman, *Adaptive Control Processes: A Guided Tour.* Princeton University Press, 1961.

[20] P. Indyk and R. Motwani, "Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality," in *Proc. of the 30th Annual ACM Symp. on Theory of Computing*, 1998, pp. 604–613.

[21] A. Hinneburg and D. A. Keim, "Optimal Grid-Clustering: Towards Breaking the Curse of Dimensionality in High-Dimensional Clustering," in *Proc. of the 25th Intl. Conf. on Very Large Databases*, 1999, pp. 506–517.

[22] P. Bakkum and K. Skadron, "Accelerating SQL Database Operations on a GPU with CUDA," in *Proc. of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, 2010, pp. 94–103.

[23] W. Liu, B. Schmidt, G. Voss, A. Schroder, and W. Muller-Wittig, "Bio-sequence database scanning on a GPU," in *Proc. 20th IEEE International Parallel Distributed Processing Symposium*, 2006.

[24] M. D. Lieberman, J. Sankaranarayanan, and H. Samet, "A Fast Similarity Join Algorithm Using Graphics Processing Units," pp. 1111–1120, 2008.

[25] Nvidia. (2018) CUDA Programming Guide, SIMT Architecture. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#simt-architecture

[26] A. Andoni and P. Indyk, "Near-Optimal Hashing Algorithms for Approximate Nearest Neighbor in High Dimensions," in *47th Annual IEEE Symp. on Foundations of Computer Science*, 2006, pp. 459–468.

[27] A. Andoni, P. Indyk, T. Laarhoven, I. Razenshteyn, and L. Schmidt, "Practical and Optimal LSH for Angular Distance," pp. 1225–1233, 2015.

[28] I. Razenshteyn, L. Schmidt, A. Andoni, P. Indyk, and T. Laarhoven. (2015-2016) FALCONN: Similarity Search Over High-Dimensional Data. [Online]. Available: https://falconn-lib.org/

[29] S. Tzeng, A. Patney, and J. D. Owens, "Task management for irregular-parallel workloads on the GPU," in *Proc. of the Conf. on High Performance Graphics*, 2010, pp. 29–37.

[30] M. Harris and K. Perelygin. (2017) Cooperative Groups: Flexible CUDA Thread Programming. [Online]. Available: https://devblogs.nvidia.com/cooperative-groups/

[31] Space Weather datasets. [Online]. Available: ftp://gemini.haystack.mit.edu/pub/informatics/dbscandat.zip

[32] Gaia DR 2. [Online]. Available: https://cosmos.esa.int/web/gaia/dr2

[33] Nvidia. (2018) Profiler User's Guide. [Online]. Available: https://docs.nvidia.com/cuda/profiler-users-guide/index.html