Noname manuscript No.
(will be inserted by the editor)

Real-time Continuous Intersection Joins over Large Sets of Moving Objects using Graphic Processing Units

Phillip G. D. Ward · Zhen He · Rui Zhang · Jianzhong Qi

the date of receipt and acceptance should be inserted later

Abstract The Multiple Time Bucket Join (MTB-join) algorithm is the state-of-the-art for processing the Continuous Intersection Join (CI-join) query over moving objects. It considerably outperforms alternatives, but still falls short of real-time application performance requirements for large sets of moving objects. In this paper, we achieve real-time performance for the CI-join query over large sets of moving objects by exploiting the computational power of commodity Graphics Processing Units (GPUs). We first analyze how the main characteristics of the MTB-join algorithm make it ill-suited to GPUs, and identify key challenges in designing efficient GPU based algorithms for the query. We then address these challenges by developing the Multi-Layered Grid Join (MLG-join) algorithm which has the following key features: (i) memory locality friendly indexing, (ii) no dynamic memory allocation, (iii) in-place object updates, (iv) lock free concurrent updates, and (v) massive parallelism. These features unleash the full potential of the memory bandwidth and parallel processing of GPUs. Furthermore, we conduct a theoretical analysis which can predict the pruning power of the MLG-join algorithm given certain parameter values used in the algorithm. This allows us to select optimal parameter values. Through extensive experimental results we show that our analysis accurately models the MLG-join algorithm's sensitivity to parameter values. The proposed MLG-join algorithm outperforms the MTB-join algorithm, and a GPU-based nested loops join algorithm, by up to two orders of magnitude, and achieves real-time performance for CI-join queries on large sets of moving objects.

Phillip G. D. Ward
Clayton School of Information Technology
Faculty of Information Technology
Monash University, Australia
E-mail: phillipgdward@gmail.com

Zhen He and Phillip G. D. Ward
Department of Computer Science and Computer Engineering
La Trobe University, Australia
E-mail: z.he@latrobe.edu.au

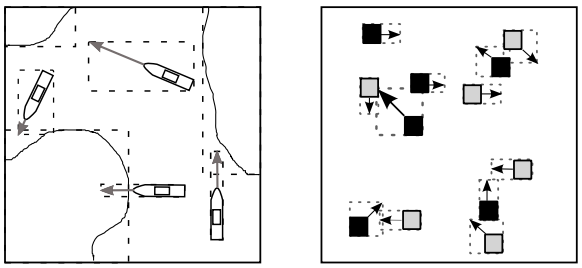
Rui Zhang and Jianzhong Qi
Department of Computing and Information Systems
University of Melbourne, Australia
E-mail: rui.zhang@unimelb.edu.au, jianzhong.qi@unimelb.edu.au

1 Introduction

The Continuous Intersection Join (CI-join) query is important for monitoring and predicting the behavior of moving objects of non-zero extents [25, 26]. It provides a continuous list of pairs of objects from two sets of moving objects that will intersect now and in the future given their current trajectories. Where object trajectories are modeled using an initial location and a linear velocity. The trajectory is updated when velocity changes or a maximum time has been reached. Forcing objects to issue updates at least once with a maximum time period allows dead objects to be removed.

Figure 1(a) shows an example of the application of the CI-join query, where shipping vessels monitor hazards such as storms [5]. Three storms and four vessels are represented by their respective Minimum Bounding Rectangles (MBRs). The CI-join query will provide a continuous report to all vessels of any storms they might encounter based on the current trajectories of the storms and the vessels. Figure 1(b) shows another application where two teams of players are in a military simulation [15] or massively multiplayer online game (MMOG) [18]. Here the query provides a continuous list of all players on two opposing teams that can in-

flict damage upon other players, both now and in the immediate future. This type of information is used in MMOGs for artificial intelligence and server-side load balancing [4].



(a) Maritime hazard monitoring (b) Military simulation/MMOG

Fig. 1 Motivating examples

The scale of problems involving CI-joins are often large (over 100,000 moving objects in military simulation [15] and 400,000 moving objects in MMOGs [18]) and require rapid object trajectory updates. With modern GPS technology, it is common for real-world objects to update their position and velocity at sub-second intervals. For real-time processing, the data structures must be updated and process queries within sub-second intervals. The moving nature of the objects distinguishes the CI-joins from traditional spatial joins and results in expensive index structure balancing as cost models and sorting become more complex. For stationary and/or point data, balancing is much simpler as objects can be pre-sorted and cost models can refer to node-overlap or velocity distributions. With moving and non-zero extent objects there is no simple order for sorting, and cost models, such as surface area of spatio-temporal volumes for the TPR*-Tree [21], are costly to compute. Furthermore, intersection checks of moving object extents over a period of time are computationally expensive, which hinders real-time query processing.

The Multiple Time Bucket Join (MTB-join) algorithm [25,26] is the state-of-the-art method for processing the CI-join query and uses two moving object index structures. It suffers from high initial cost for building the index structures and high maintenance cost of keeping the two index structures up-to-date as object trajectories change. This limits the scalability of the MTB-join algorithm. Zhang et al. [25,26] report that the MTB-join algorithm requires several seconds to process a CI-join on two sets of 100,000 objects using today's regular commodity hardware, which is slower than sub-second response time required in real-time applications.

In this paper we propose to leverage the computational power of Graphics Processing Units (GPUs) to achieve real-time processing of the CI-join query on large sets of moving objects. In order to use GPUs efficiently, a job must be broken up into thousands of small pieces which can be performed in parallel with minimal sychronization and load imbalance. Therefore a natural solution is to use a brute force join algorithm such as the nested loops join. However, such a solution will not yield satisfactory performance due to the huge number of comparisons (one for each object pair) to find intersecting objects.

Indexes can be used to prune the number of intersection checks. However, most existing indexes [12,19–21] require the number of objects in tree nodes or grid cells to shrink and grow in response to object updates, leading to expensive dynamic memory allocation. In addition, concurrent updates on these indexes require locking or atomic operations. Both dynamic memory allocation and locking are expensive on GPUs. This means the MTB-join algorithm is ill-suited to GPUs since it uses TPR-tree indexes which requires frequent dynamic memory allocation and locking operations [19].

The above discussions point out the design challenges facing an efficient GPU based algorithm for the CI-join query. We propose a novel algorithm named the Multi-Layered Grid join (MLG-join) which addresses those challenges. The MLG-join algorithm groups objects into partitions based upon memory locations, and indexes each partition in a separate layer of a multi-layered grid. As a result, data is accessed at the partition grain via the grid index. Storing objects within each partition consecutively allows index look-ups to take maximum advantage of the massive GPU memory bandwidth[1]. In contrast to existing tree/grid based structures, our multi-layered grid approach performs concurrent updates without using dynamic memory allocation or atomic operations. We avoid dynamic memory allocation by reserving a separate bit for each partition within each grid cell. We avoid locks and atomic operations by handling insertions as writing a bit value of 1 into the grid, regardless of the previous state of the bit. Deletions are handled by periodic rebuilding of the grid, which maybe done highly efficiently thanks to the great parallelism provided by GPUs. This allows for thousands of index updates, and queries, to be executed in parallel.

In summary, the MLG-join algorithm has five key features: (i) memory locality friendly indexing, (ii) no dynamic memory allocation via a multi-layered grid structure, (iii) in-place object updates, (iv) lock free concurrent updates and (v) massively parallel maintenance and querying. These features overcome the drawbacks of the previously discussed approaches. Although single level partition indexing is coarse, experiments

[1] Currently up to 192GB/s for the NVIDIA GTX 680.

show that the multi-layered grid can prune 99% of intersection checks in typical workloads. The coarse granularity also eases load balancing as each partition has a fixed size. Experiments show that the grid can be rebuilt within 100ms for 500,000 objects, avoiding the expensive setup cost of the initial join process in the MTB-join algorithm.

It is worth noting that some of our ideas are not limited to the CI-join query, and are applicable to large batches of spatial queries such as the range, similarity search [3] and kNN queries [17]. Our key contributions are as follows.

- We propose a novel high-performance GPU based solution to the CI-join query called the MLG-join algorithm. We overcome the challenges of load balancing, dynamic memory allocation and high maintenance costs of moving object indexes on GPUs by indexing partitions (groups of consecutively stored objects) in a fixed-sized, multi-layered grid. The grid is updated in parallel without locking or atomic operations.
- We theoretically analyze the percentage of object intersection checks that have been pruned by the MLG-join algorithm. The analysis is experimentally validated and is used to guide algorithm parameter selection.
- We conduct an experimental study comparing the MLG-join algorithm against the state-of-the-art CPU-based approach (MTB-join algorithm) and a naive GPU solution (GPU nested-loop join algorithm). The results show that the MLG-join algorithm outperforms both alternative algorithms by up to two orders of magnitude. The MLG-ioin algorithm achieves real-time processing of the CI-join query on large sets of moving objects using a regular GPU equipped commodity computer.

The rest of the paper is organized as follows. Section 2 gives the problem definition and background informations on GPUs. Section 3 reviews related work. In Section 4, we present the MLG-join algorithm and its implementation on the GPU. Section 5 provides a theoretical analysis for choosing algorithm parameters. Section 6 presents the experimental setup and results, and finally Section 7 concludes the paper.

2 Preliminaries

In this section we provide a problem definition and describe the performance characteristics of GPUs.

2.1 Problem definition

The CI-join query was originally proposed and defined by Zhang et al. [25]. The query assumes moving objects with non-zero extents bounded by MBRs, modeled using linear functions of time [22]. The intuition here is that the velocity of an object usually stays unchanged for a short period of time. As a result, we can model the movement of an object by its current location and velocity rather than keeping track of every location that the object passes through.

Objects are requested to report their locations whenever their velocity changes or a maximum update interval (T_M) is about to expire (often referred to as a *heartbeat*). If an object has not reported its location within T_M then we assume the object has left the system. The CI-join query is then formally defined as follows:

Definition 1 *Let A, B be sets of moving objects with non-zero extents, and let $Mbr(x,t)$ be a function that returns the MBR of an object x at timestamp t. Let $a \in A$, $b \in B$. The* ***continuous intersection join (CI-join)*** *query finds every pair (a,b) such that $Mbr(a,t) \cap Mbr(b,t) \neq \emptyset$, for any time $t \in [t_q, \infty)$, where t_q denotes the timestamp when the query is issued.*

The CI-join query is computed in two phases, generating the initial join pairs (initial join) and then maintaining the join pairs continuously as objects are updated (maintenance).

When a query is issued, the initial join phase begins. We compute every pair of objects that intersect now and/or in the near future (up to $t_q + T_M$, where T_M denotes the maximum update interval), and the time period that a pair of objects stays intersecting. This computation is based on the current locations and velocities of the objects. Its results form a list L of tuples in the form of $\langle a_i, b_j, t_s, t_e \rangle$, where a_i and b_j denote two intersecting objects and t_s and t_e denote the starting and ending timestamps of the intersecting period. We then start the maintenance phase. At every timestamp t, we just need to recompute the join result (up to $t + T_M$) for the objects that have reported updates at t, and update the list L accordingly. We then report the object pairs in the list whose intersecting periods overlap t. This way, we obtain the intersecting pairs for every timestamp while constraining the computational cost by avoiding the join on every object at every timestamp. In this paper we study how to process the two phases using the GPU efficiently.

We assume that the CI-join query will be performed entirely in memory, since the size of both the main memory of the system and device memory of GPUs are currently on the order of several GB. Indexing and

storing two sets each containing 500,000 objects is on the order of several MB.

2.2 Graphics Processing Unit (GPU)

Although the GPU is powerful, using it to accelerate the processing of the CI-join query is difficult. Understanding the main performance characteristics of the GPU is key to understand the difficulty and opportunities it offers.

Table 1 provides a summary of the differences between the CPU and GPU used in our study. The main difference is that the GPU has many simple cores whereas the CPU has a much fewer number of powerful cores. GPUs simple cores lack out-of-order execution and branch-prediction. The benefit of GPUs stem from lightweight context switching, high data throughput and the ability to execute thousands of threads at once. Coordinating the thousands of GPU threads is difficult as high speed communication can only occur between predefined thread groups called blocks.

Locking and atomic operations are particularly expensive on GPUs because the way to achieve performance improvement on the GPU is through massive parallelism. According to the well-known Amdahls law even a very small amount of sequential code (caused by locks or atomic operations) can very dramatically limit the speedup achieved from the massive number of parallel threads. This is reason we have designed our algorithm to avoid any use of locks or atomic operations.

Threads within a block are processed in batches, called warps; all threads in a warp must execute the same instruction at the same time. Any branch divergence (caused by predicate evaluation) violates this and has a multiplying negative effect on execution time as some threads sit idle while others evaluate predicated statements. We quantify the idle time using a metric called *occupancy*. At full occupancy, all cores are being used by threads at all times. Low occupancy means some physical cores are idle. Thus is the result of too few threads running, or threads that cannot operate in parallel due to divergence.

GPU threads operate on a separate memory block (device memory). The cost of moving data between main memory (for CPU processing) and device memory (for GPU processing) is high, and can be the most expensive step in hardware accelerated applications.

Device memory access has high latency and high bandwidth. Thousands of parallel threads can be used to hide these latency costs. Consecutive threads are required to request data from consecutive blocks in order to achieve full throughput. This is known as *coalesced* memory access. Any random (*non-coalesced*) memory access will result in reduced bandwidth. Thread blocks have a small dedicated explicit cache called *shared memory*, and an implicit cache. Both the shared memory and implicit cache have much lower access latencies than device memory and can be used to mitigate non-coalesced access penalties.

Brute-force algorithms are highly competitive solutions on the GPU, due to minimal inter-thread communication, predictable memory access patterns and simple execution paths. One such algorithm is the *nested-loops join*, where every object in set A is compared with every object in set B. The outer loop can be executed in parallel across thousands of threads. This magnitude of threads is required in order to hide the latency of device memory access. Full use of a GPU computational potential can easily be achieved as each pair evaluation can be performed in parallel, accessing contiguous memory locations and performing the same set of instructions. Although the nested-loops join is easy to parallelise, it incurs too much computational cost by comparing every pair of objects from the two sets without pruning. Complex algorithms are more difficult to parallelise effectively. Conforming to the operational requirements of GPUs is challenging, and crippling for most algorithms.

In summary GPUs are characterized by the following key attributes: 1) a large number of threads operating in parallel across hundreds of cores, 2) high memory bandwidth for coalesced (sequential) access, high memory latency for non-coalesced (random) access, 3) limited cache and shared memory, 4) heavy penalties for branch statements, and 5) minimal inter-thread communication.

3 Related Works

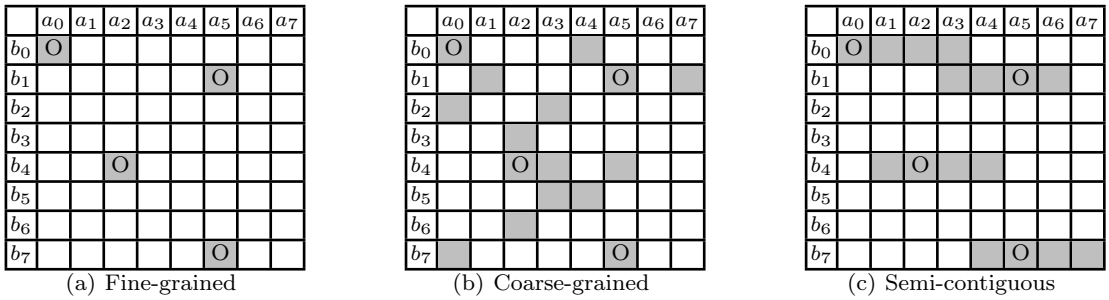
In this section we describe four areas of related work: indexing moving objects, existing algorithms for processing CI-join queries, join processing on GPUs and main memory indexing structures on CPUs and GPUs.

3.1 Indexing moving objects

The Time Parameterized R-tree (TPR-tree) [19] is a popular moving object index which is an extension of the R-tree [9] into the temporal domain. The TPR-tree stores an object's position at a reference time and their associated velocity at that time. Each MBR in a TPR-tree has an associated Velocity Bounding Rectangle (VBR) which bounds the velocities of the objects bounded by the MBR. The TPR*-tree [21] improves over the TPR-tree by offering improved insertion and

	Num. Cores	Frequency (GHz)	Num. Transistors	BW (GB/sec)	Peak SP Flops (GFLOPS)
Core i7-950	4	3.06	0.73B	25.6	97.92
GTX 460 SE	288	1.30	1.95B	108.8	748.8

Table 1 Core i7-950 and GTX 460 SE specifications. BW: local DRAM bandwidth, SP: Single-Precision Floating Point



(a) Fine-grained (b) Coarse-grained (c) Semi-contiguous

Fig. 2 Examples of remaining search spaces from three different indexes. The grey squares and white squares represent remaining and pruned comparisons, respectively. Intersections are marked with letter O.

deletion algorithms. Although the TPR-tree and TPR*-tree offer fast query times, they both suffer from high initial building cost and high maintenance costs.

Other moving object indexes, such as the B^x-tree[12], offer lower build and maintenance costs but still require dynamic memory allocation and locking for concurrent updates, both of which are costly on GPUs. Ali et al. [1] build an index on wavelets to reduce the workload of continuous retrieval of 3D objects. It is based on 3D object modelling and does not apply to 2D objects straightforwardly.

3.2 MTB-join algorithm

The only existing work on the CI-join query was done by Zhang et al. [25,26], for CPUs. To our knowledge there is no existing work on processing the CI-join query on GPUs. Zhang et al. use two TPR-trees (one for each set A and set B) to prune the the number of comparisons during join processing. Over time TPR-trees degrade in performance due to the expanding of MBRs. Even with regular updates, any future looking query requires expansion.

Zhang et al. reduced the computational load of the CI-join query by imposing a time constraint, observing that a join result between any two objects only needs to be valid until the next update on either of the two objects. The heartbeat (T_m), the maximum time between updates for any object, was adopted as the time constraint. This lowered the computation cost of queries, but increased the need for query updates as the result accuracy was also time constrained.

To overcome the high frequency of updates the MTB-join algorithm was proposed. Query updates were optimized by dividing objects into time buckets based upon object update behavior. Each time bucket was represented by a TPR-tree.

Although the MTB-join algorithm uses some very clever techniques to get the most out of the TPR-tree for pruning the dataset, it is fundamentally unsuitable for use on the GPU due to the branching nature of tree traversals. In addition the overhead of frequent tree updates caused by object updates is particularly expensive on GPUs as they required random manipulations of memory.

3.3 GPU-based join on static objects

He et al. [10] proposed GPU counterparts for the common relational database join algorithms including nested-loops join, indexed-loops join, sort-merge join and hash join. An emphasis was placed upon the usage of bandwidth via coalesced access and shared memory. Their work resulted in speed-ups of between 2 and 27 fold for the primitives and, 2 and 7 fold for the joins. Our work differs from theirs by dealing with a *continuous* join query over *moving* objects.

Bandi et. al. [2] integrated the use of GPUs inside the Oracle9i commercial database for processing queries on stationary spatial objects. In particular they processed the spatial join using two techniques, an indexed nested-loops join using an R-tree and a hash styled join using a quadtree. Their work was based upon stationary objects and the performance benefits are not applicable to moving objects due to the cost of processing updates. The idea of this GPU based join algorithm can also be used in emerging location based applications such as location selection [11,16], which essentially performs join operations on spatial indexes to find the optimal locations.

Bohm et. al. [3] demonstrated the use of GPUs for processing complex data mining tasks. A building block of this work included a GPU based similarity search algorithm. The approach had a significant speed improvement over the CPU counterpart and was an exam-

ple of an indexed nested-loops join using a tree structure. This approach could be extended to perform intersection checks between two sets but there is difficulty extending it to moving objects with non-zero extents. Firstly, this work relies upon sorting the data before building the index. This is difficult as the data of the CI-Join is dynamic and changes regularly, therefore requiring frequent re-sorting under this scheme. Secondly, this work is designed for point data with no obvious mechanism for handling objects with extents. Indexing ranges in a tree structure for GPU access is a difficult task as efficiency requires that each query must traverse the same depth (number of nodes visited). However, since ranges can span multiple tree nodes, it is difficult to predict the traversal depth. Further problems arise when the data is dynamic and cannot be intensively processed offline.

The papers by Zhang el al. [23,24] propose a grid-based approach to perform fast point in polygon (PIP) tests on the GPU. Our approach is similar to theirs in that we have a pruning stage like their filtering stage and we both use a grid-based approach on GPUs. However, their filtering stage indexes both sets of data. They also index each polygon separately (each polygon is assigned a separate id) in the grid. In contrast we only index one set of data, the partitions of random objects (objects that just happen to be next to each other in an array) in a grid. Our partitions are fundamentally different from their polygons, since we effectively treat the entire partition of separate objects as one big object whose area is composed of the union of all of its constituent objects. This effectively means we assign a single identifier (id) for a whole partition of objects. In contrast they index each polygon separately onto the grid and thereby assigning a different id to each polygon. The drawback of the their approach is they need to store and therefore query a much larger grid since each grid cell will contain all identifiers of all the polygons that intersect the grid cell. In contrast for each grid cell we only store a single bit per partition of objects. Our indexed data structure suits the architecture of the GPU much better, because our grid has a fixed maximum length (one bit per partition) and thus does not require dynamic memory allocation or a dynamic structure (linked-list), and memory requirements are more predictable. A much larger maximum fixed length grid, or a dynamic grid (e.g. linked-list of ids) would be required for a unique id for each object, rather than just each partition.

Furthermore, by indexing both sets of objects, Zhang et al. [23,24] must perform a scan to detect intersections and then remove duplicates. The MLG-join algorithm only indexes one set as partitions, and queries (probes) this index with the other set. Any resulting duplicates are intrinsically removed due to the bit-wise OR operations which occur during the probe.

In addition to the above differences the solution by Zhang el al. [23,24] is only designed for a static join whereas we are doing a continuous join. They do not need to worry about maintenance. We are intersecting two sets of objects with rectangular extents (4 sides) whereas they are intersecting a set of points against a set of polygons (any number of sides).

3.4 CPU and GPU main memory index structures

Kim et. al. [14] have developed FAST, a highly optimized CPU and GPU implementation of a binary tree. This implementation is highly suitable to GPUs and demonstrates considerable performance benefits. Unfortunately it is not suitable for two-dimensional range data (object extents) and tree updates are costly. Updates are performed in batches to reduce this cost, but the memory locations are unpredictable resulting in expensive random writes on GPUs.

Sidlauskas et. al. [20] developed the PGrid, a grid-based indexing technique for parallel processing of moving object queries and updates. This implementation relied upon atomic SIMD operations and dynamic memory allocation, both of which are costly on GPUs. In contrast, our multi-layered grid does not use any atomic operations or locks to support concurrent updates. In addition, our approach does not need dynamic memory allocation and allows for in-place updates without moving objects between grid cells.

A grid based index is much simpler to update than a tree. Updating an object involves removing the object from the cells it previously occupied and inserting it into the cells that it now occupies. However, since there can be a variable number of objects per grid cell, this simple operation still needs a dynamic data structure, such as a linked list.

We overcome these hurdles by implementing a grid based structure of fixed size that indexes data at a coarse granularity. This eliminates the need for dynamic memory allocation which greatly reduces the maintenance costs.

4 Multi-Layered Grid Join (MLG-join)

In this section we first describe the high level objective of the MLG-join algorithm. Then we present the detailed steps of the algorithm and its GPU implementation.

We use the indexed loops join approach to process the CI-join query. Alternative approaches, such as the

sort-merge join and hash join, have difficulty processing objects with non-zero extents. Furthermore, sorting poses additional challenges with moving objects and dynamic data, especially for GPU processing.

The indexed loops join approach operates as follows. At the highest level the algorithm first uses an index to prune some comparisons (object intersection checks). We call the set of comparisons remaining after pruning the *remaining search space*. Second, it performs a nested-loops join for the remaining search space. Different index structures can prune the search space by different amounts.

Figure 2 describes three possible remaining search spaces after pruning using different hypothetical indexes. We assume that objects a_0 to a_7 are stored contiguously in memory. Figure 2(a) is the result of fine-grained pruning, only the intersecting pairs remain (4 comparison operations). Figure 2(b) is the result of coarse-grained pruning (16 comparison operations remain). Figure 2(c) is the result of coarse-grained contiguous pruning, the remaining comparisons are *semi-contiguous* in memory (16 comparison operations).

An index that can achieve Figure 2(a) is clearly the most desirable if we are solely interested in having the least remaining comparisons. However, in order to achieve this level of pruning we need to index objects at a very fine grain. The consequence of fine grained indexing is high update costs since it means the index is very sensitive to the exact location of objects. To reduce the update cost, a coarser grained index can be used, such as in Figures 2(b) and 2(c).

On GPUs, Figure 2(c) is more desirable than Figure 2(b). The reason is that the contiguous nature of the objects in the remaining search space allows the comparisons to be performed in parallel with coalesced memory access. Using the parallel architecture of GPUs, the coarsely pruned search space of Figure 2(c) can be examined in approximately the same time as Figure 2(a), despite having four times the number of remaining comparisons. Thus on GPUs we can avoid using a fine grained index and its accompanying high update costs without significant query performance impact.

The crux of the MLG-join algorithm is coarse-grained, semi-contiguous pruning. The remaining search space consists of blocks of thousands of contiguous comparisons, which are then processed on thousands of GPU threads in parallel with fully coalesced memory access. We develop an indexing structure to produce a contiguous pruned search space, which can handle moving objects without sorting. This structure allows for thousand fold parallelism on GPUs when building, querying and updating, without locking mechanisms. This is achieved by indexing groups of objects (partitions) which are contiguous in memory and performing all queries, and updates for a single partition in parallel.

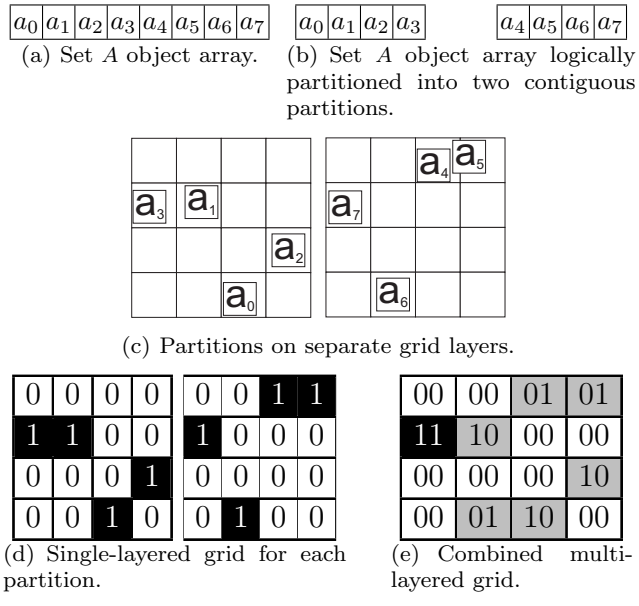
4.1 MLG-join algorithm

For ease of understanding we first present the sequential version of our MLG-join algorithm and then describe the parallel GPU version in Section 4.2.

The MLG-join algorithm uses **two data structures**. The first is the object array which stores the position, velocity, extent and update information for each object. Set A and set B objects are stored in separate object arrays. The second is a multi-layered grid which coarsely indexes set A objects.

The MLG-join algorithm processes the CI-join query in the following four steps with set B functioning as a batch of range queries on the multi-layered grid of set A.

1. *Construction*, where the grid layers are built.
2. *Probing*, where the grid layers are queried.
3. *Stream preparation*, where the query results are transposed to create a reduced search space.
4. *Intersection*, where the search space is processed.



(a) Set A object array.

(b) Set A object array logically partitioned into two contiguous partitions.

(c) Partitions on separate grid layers.

(d) Single-layered grid for each partition.

(e) Combined multi-layered grid.

Fig. 3 Constructed multi-layered grid. Shading based upon the proportion of partitions occupying the cell none (white), one (grey) and two (black).

4.1.1 Construction step (Building the index)

This step involves indexing set A objects using a multi-layered grid. The Set A object array is logically partitioned, meaning that the object array is not physically changed but rather the objects within each logical partition are indexed in a separate layer of the multi-layered grid.

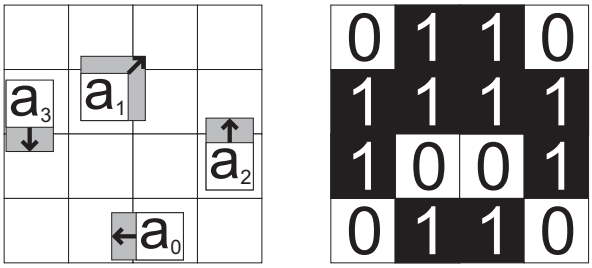
The objects are separated into partitions based on **memory locality (location in the sequential array)**. This differs from most indexing structures which group objects based on position and/or velocity. The benefit is two fold. Firstly, it helps to produce a contiguous search space after pruning. Secondly, object location and/or velocity updates can be done in-place without moving objects between partitions.

Figures 3(a) and 3(b) depict logically partitioning set A objects into two partitions. Note the objects in the set A array are not moved during the partitioning but rather as mentioned above the partitioning is logical instead of physical. The logical partitioning allows us to create a separate grid layer for each partition. Figure 3(c) shows the set A objects of each partition placed on a separate grid layer. It should be observed that the objects in set A are randomly ordered. Furthermore their indices have no deliberate correlation to spatial information such as position or velocity. This results in the contents of a partition, p, being randomly assigned in regards to spatial information. It should be noted, in practice each partition has thousands of objects.

A grid layer is analogous to an overhead silhouette of the objects within one partition. A white grid cell means the cell is empty; a black grid cell means the cell is occupied by at least one object of the partition. Occupation is defined as an object overlaying any part of a cell. As shown in Figure 3(d), a white cell is represented by 0 and a black cell is represented by 1. In this example the layers are combined into a two-layer grid (one layer per partition) by concatenating the bits of like grid-cells to make a bit-string, as shown in Figure 3(e).

The multi-layered grid does not assume there is any spatial clustering present in each partition. Despite this fact it is still very effective at pruning because the full spatial coverage of any one partition is typically very small compared to the full space. This is true even if the objects are distributed randomly. Experimental results show that our index can prune more than 99% of comparisons when the default number of parameters is set to 128. Furthermore, for skewed data our random assignment of objects into partitions actually assists with load balancing since the high-density areas will be spread across multiple partitions. In contrast, other indexes, e.g., R-tree, will struggle due to the high number of overlapping nodes in the high-density areas.

Moving objects. In the literature, the R-tree has been extended with a time parameter to cater for moving objects. This has made the MBR of each object a volume representing the MBR over a period of time. This would require a 3D grid in our case. A 3D grid would increase the memory required for our grid, along with the computational cost of constructing and probing it.



(a) MBRs of moving objects

(b) Gird cell representation of MBRs

Fig. 4 Grid representation of object motion.

To avoid the extra costs, we enlarge each object's MBR to include all space that the object covers over a period of time (ΔT). Figure 4(a) shows the MBR of the objects (a_0,a_1,a_2,a_3) from the *current time* to *current time* + ΔT. Figure 4(b) shows the grid representation of the object. In Section 4.3 we discuss how the value of ΔT is determined.

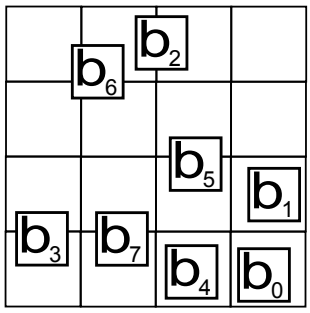
4.1.2 Probing step (Querying the index)

This step involves the other set of objects (set B) querying the grid. The query generates a bit-string for every object b in set B, where a bit represents whether b intersects at least one object in a partition of set A (1 for yes and 0 for no). A zero in an b's bit-string means b does not intersect any object in the partition corresponding to that bit.

For example, suppose we have a set B as shown in Figure 5(a) probing the multi-layered grid produced in Figure 3(e). Then the output bit-strings are shown in column two of Figure 5(b). As we can see from Figure 5(b), the remaining search space represented by the bit-strings are semi-contiguous, which satisfies our algorithm design objective as discussed at the beginning of the section.

4.1.3 Stream preparation step (Results transpose)

This step involves converting the probe output into a stream of set B objects associated with the partition that these objects intersect. The result is a list of set B objects associated to an intersecting partition computed from the probe output. An example of the stream corresponding to the probe output of Figure 5(b) is depicted in Figure 5(c). This step is very similar to a transpose operation, where the probe output is considered a matrix with a column for each partition and row for each object.



(a) Set B objects acting as queries on grid.

		p_0				p_1			
	Output	a_0	a_1	a_2	a_3	a_4	a_5	a_6	a_7
b_0	00								
b_1	10			O					
b_2	01					O			
b_3	00								
b_4	10	O							
b_5	00								
b_6	11		O						
b_7	01							O	

(b) Probe output

Partition	Bit-string	Stream
p_0	10	b_1, b_4, b_6
p_1	01	b_2, b_6, b_7

(c) Stream

Fig. 5 Example probe and stream preparation.

4.1.4 Intersection step

This step processes the intersection checks on the remaining pairs. The intersection check takes into account the precise spatio-temporal location of the objects and checks if an actual intersection occurs. If the intersection check detects an intersection, the time period of the intersection (t_s, t_e), is computed. This is similar to three nested-loops. The outer-most loop is the partitions. The next loop is the objects in a partition. The inner most loop is the set B objects in the stream, and the operations of the inner most loop are checking whether two objects intersect each other. The workload of this nested-loop is close to the size of the solution set, due to the pruning power of the MLG-join algorithm as validated by the experimental study. We summarise the above algorithm steps in Algorithm 1.

Algorithm 1: The MLG-join.

input : Two sets of objects A and B. Set A divided into a set of partitions P.
output: Set of tuples S containing a, b, t_s and t_f where a is an object from set A, b is an object from set B and t_s, t_f refer to the start and end time of an interval where objects a and b intersect.

```
   /* Project a's silhouette onto grid for p          */
 1 for p ∈ P do
 2 |   for a ∈ p do
 3 |   |   ConstructMultilayeredGrid(p, a)
   /* Create probe output for b from grid             */
 4 for b ∈ B do
 5 |   Probe[b] ← ProbeMultilayeredGrid(b)
   /* Convert probe output to stream (list)           */
 6 for p ∈ P do
 7 |   for b ∈ B do
 8 |   |   if p ∈ Probe[b] then
 9 |   |   |   List[p] ← List[p] ∪ b
   /* Test each b in stream vs. each a in partition   */
10 for p ∈ P do
11 |   for a ∈ p do
12 |   |   for b ∈ List[p] do
13 |   |   |   S ← TestIntersection(a,b)
```

4.2 Parallel query execution of MLG-join on GPUs

First we describe how the MLG-join algorithm can be implemented on GPUs without locking mechanisms, followed by a detailed explanation of the implementation.

In general, the input for each step of the algorithm is read-only removing the need for locking, and the output for each thread has its own memory. The exception to this is the grid during construction. Each thread must share the grid. Fortunately, the construction process is commutative, i.e., the iterations may be performed out of order and the result will be the same. There is no need for locking or atomic operations as any conflicting parties will be trying to do the same thing, toggle a bit from 0 to 1. Although bits are accessed at 32 bit integer grain, no lost updates occur. This is because each cell is stored as a separate integer, and each partition accesses the grid sequentially. The result being that the only time two threads attempt to access the same integer, they will both be attempting to access the same cell, on behalf of the same partition, and setting the same bit to 1. Although threads which are in different warps will not be synchronised for this operation, the result will still be accurate as long as both threads are attempting the same operation (toggling the same bit to 1). Due to the memory hierarchy on the graphics card, the two requests will filter up through local memory and cache, to the global memory controller. When this occurs, the two operations will either be combined, or executed in serial. As no thread will ever be setting a bit to 0, and no two different partitions are executed at the same time, the operation will achieve the correct result despite the order of execution being unpredictable.

The method for parallel execution on GPUs is described in the following paragraphs. Each step of the algorithm is performed using a different kernel.

In the **construction step**, thread blocks are assigned to each partition, with multiple blocks provided if the number of objects in a partition exceed the max-

imum number of threads in a block. Threads are assigned consecutively to objects, with one thread per object. An occupancy penalty can occur when the number of objects is significantly smaller than the maximum number of threads in a block, resulting in under-use of the hardware. This penalty can be minimised or completely avoided through parameter selection. The first step is to read the attributes of each object. As the objects in adjacent threads are in adjacent memory positions, the memory access is coalesced. This makes maximal use of the available bandwidth which is the motivation behind the partition structure. The next step is to calculate which grid cells should be modified. This process is identical for each thread, achieving full occupancy on GPUs. The next step, writing to the grid, is not coalesced for each partition, but is coalesced for each object. This is due to adjacent objects in memory not residing in adjacent grid cells, but grid cells spanned by a single object being adjacent in memory. In the scenario where an object only spans a single grid cell the performance penalty is small for this potentially non-coalesced operation as the total cost of the operation is only a few bits per object. In scenarios where objects overlap many grid cells, the memory cost of this operation increases whilst becoming increasingly coalesced. This behavior is controlled through grid cell size, a tunable parameter.

In the **probe step**, a thread is assigned to each object in set B sequentially. Reading each objects' attributes is a fully coalesced process. Calculating which grid cells to read from achieves full occupancy as in the construction step. Reading from the grid is a non-coalesced operation, but the cost is minimal. The read bit-strings are processed into a single bit-string using OR operations. This is identical for each thread and therefore achieves maximum throughput. The final step is writing these strings into memory. As the strings have a fixed length (1 bit per partition), an array may be allocated prior to this step. Writing to this memory is entirely coalesced and achieves maximum bandwidth.

In the **stream preparation step**, a thread is assigned to each partition. Each block reads the probe output of a single object in set B into shared memory. Each thread then checks if the bit corresponding to its partition is set to 1. If it is, the ID of the object is written to that partitions' stream. Full memory bandwidth can be achieved during the read step by loading a batch of objects from set B at a time into shared memory. Full occupancy is achieved by the threads as the operations required to check each bit are the same. This process is repeated for each object in set B.

A difficulty arises when adding objects to a stream, which involves writing to memory randomly. This random write is minimized by only storing object IDs in the stream.

The reduced search space is all that remains to be processed. This remaining workload is highly suitable for GPUs as the data is mostly contiguous, which lends itself to coalesced memory access patterns. The operation to be performed on this data is an intersection check which will be identical computationally for each object pair, resulting in zero branching. This all results in the potential for thousands of comparisons to be processed in parallel.

In the **intersection step**, the threads are allocated in an identical fashion to the construction step. Each block corresponds to a single partition. Reading the object attributes is highly coalesced as stated. The next step is to process the stream of objects. Each object in a partition uses the same stream, therefore each thread in a block requires the same objects from set B. This can be exploited by loading all the objects within the stream into shared memory.

Following this, each thread in the block checks for an intersection between its own object and the objects in shared memory, and returns the time period that the two objects intersect for, if any intersection occurs. This step is the most computationally intensive of the algorithm. In implementation, it not only achieves full throughput via each thread performing the same operations, it also performs less memory operations as each object in the stream is only read once by the entire partition. The number of memory operations per block is one for each object in the partition (coalesced), **plus** one operation for each object in the stream (non-coalesced). Without exploiting the thread allocation, this computation would require one operation for each object in the partition (coalesced), **multiplied** by one operation for each object in the stream (non-coalesced). This provides a significant reduction in the number of global memory operations required, which in the best-case is equivalent to the number of objects per partition.

Handling skew. A potential performance issue with many GPU algorithms is handling skew. The above implementation relies upon similar workloads being undertaken by each partition. The partition approach intrinsically handles skewed datasets, due to the fact that objects are grouped based on memory locality instead of spatial locality. This makes it unlikely that each partition will have a significantly different workload. If an area of the grid has a higher density, which is possible in most applications, this density is likely to be spread across multiple partitions.

4.3 Maintenance of continuous query results

Join maintenance is performed in two steps. First the object array and grid are updated, and then the join results are updated by re-probing the updated grid. We will consider updates in batches as it is more efficient and an individual object update is a special case of a batched update.

In-place object updates. Objects are updated in-place within the object arrays. Objects from set A also insert a new projection of themselves onto the grid. The old projection of the object is not deleted since it is impossible to reverse an inserted projection. This does not affect the correctness of the join result as it can only cause false positives, not false negatives, which are removed in the intersection step. We periodically remove all stale projections by clearing and rebuilding the grid (detailed later in this section).

New objects are inserted at the end of the object array. If the object is a set A object then it is also inserted into the last partition. If many insertions occur, an extra partition, and corresponding grid layer is set up for them. The last partition and any new partitions then run through a construction step to populate their respective grid layers. *Deleted objects* are set to null in their object arrays.

After updating the data structures, the probe result is updated as follows. All set B objects probe the grid layers of updated set A partitions (set A partitions that have an object update). This is computed by only probing the final (updated) grid layers. Then the updated and newly inserted set B objects probe the grid layers of all set A partitions. A stream is prepared and processed. This is performed in the same way as a regular probe and preparation step, the difference being a smaller set B. New join results are added to the existing result set and any deleted objects are removed.

Grid maintenance. The grid must be periodically cleared and rebuilt, removing the projections of deleted and out-of-date objects to ensure it retains its efficiency and correctness. We call the time period between grid rebuilds ΔT. As ΔT increases, more updates occur between rebuilds resulting in more stale projections on the grid. The size of projections also increase as ΔT increases, resulting in more expensive construction and probing of the grid. A smaller ΔT means the stale data is removed more often, making the grid more efficient but resulting in more frequent rebuilding.

Zhang et al. [26] thoroughly investigated how index rebuilding time offsets the join performance when analysing optimal parameters for the MTB-join. Following the results of their study, we set ΔT to the heartbeat time (T_m). Every object must update every T_m time period, therefore at the end of T_m, every object in the grid will have stale data.

Zhang et al. [25] proposed grouping objects into time buckets based on similar update patterns and indexing each bucket individually so as to reduce index rebuilding cost. We use a similar approach, i.e., grouping partitions into time buckets and offsetting their rebuild times. E.g., given 2 buckets, the first would be rebuilt at ΔT, and $2\Delta T$, etc., and the second would be rebuilt at $\frac{3}{2}\Delta T$, and $\frac{5}{2}\Delta T$, etc.

The multi-layered grid for each bucket is stored separately. We choose a multiple of 32 for the number of partitions (layers) per bucket. This simplified the stream preparation and the OR operations in the probing step. The bit-string in the stream preparation step would be stored as a series of 32-bit integers, which is well suited for GPU memory layout. Thus, it determines the number of partitions per bucket.

4.4 Memory Usage Analysis

There are two main areas within MLG-join where memory usage can be large. The first is the size of the multi-layered grid and the second is the temporary memory used by the transpose operation performed during the stream preparation step.

A simple way to reduce the size of the multi-layered grid is to create a coarse-grained grid (large grid cell size and small number of partitions). It turns out a relatively coarse-grained grid is actually desirable for both performance and memory usage because a finer-grained grid results in too many separate memory reads and writes. In our experiments the default setting of 1000x1000 grid cells, 256 partitions and grid cells of 0.5x0.5 gives optimal performance and the memory usage is only 64MB. Even in scenarios, which consist of much larger worlds, many more objects, it is still better to use coarser-grained implementations due to the selectivity.

For the stream preprocessing step it is difficult to perform the transpose operation in-place and therefore a large temporary buffer is needed. Modifying the logistics of the algorithm to operate in a memory-constrained environment can circumvent this limit. Such a modification can be done without impinging on processing time, through pipelining. This is left as a direction for future work.

5 Theoretical Analysis

In this section we first present a cost model for the MLG-join algorithm which includes the key parameter

X, the set of object pairs remaining after the pruning steps (first three steps) of the join algorithm. The parameter represents the effectiveness of the grid at reducing the number of object intersections to check in the intersection step. Next in Section 5.1 we will quantify the effect that the number of partitions (α) and grid cell width (γ) have on the size of X ($|X|$).

The MLG-join algorithm's performance can be analysed by examining the computational cost, O, required for each of the four steps individually.

$$O = k_1|A|\frac{xy}{\gamma^2} + k_2|B|\frac{xy}{\gamma^2} + k_3|B|\alpha + k_4|X| \quad (1)$$

Here x and y are the *width* and *height* of the objects respectively. For non-stationary objects, x and y represent the enlarged MBR depicted in Figure 4. Given that the computational cost of each different operation is unique, e.g., calculating the intersection of two rectangles is more costly than reading a bit, a step specific cost coefficient is introduced, where k_i is the coefficient for step i.

The first part of the equation corresponds to the construction step, where each object in set A performs an operation for each grid cell it occupies. This is calculated as the ratio between object size and grid cell size.

The second part of the equation corresponds to the probing step and is similar to the construction step, with reading being performed instead of writing. Thus it has the same number of operations as the construction step (but proportional to $|B|$ instead of to $|A|$).

The third part of the equation corresponds to the stream preparation step, where each bit of each bit-string is analyzed. The number of operations is proportional to the length of the bit-string (α) and the size of set B.

The fourth part of the equation corresponds to the intersection step, where each object pair intersection not pruned in the first three steps is checked.

It is obvious the first three steps will scale linearly with set sizes. The unknown is X, which we derive as a function of α and γ. This function quantifies the benefits of the pruning steps (steps 1 to 3) of the MLG-join algorithm given a particular partition size and grid resolution.

5.1 Analysis of pruning power of the multi-layered grid

The effect the number of partitions (α) and grid cell width (γ) have upon X is important. A naive assessment of the algorithm suggests a higher resolution grid (smaller γ) and more partitions (higher α) will result in more pairs being pruned (smaller X).

A limitation is placed upon γ, in that the smaller it is, the more grid cells must be read and written in the probe and construction steps. Furthermore, smaller values of γ will increase the grid size in memory. Thus the cost of a higher resolution grid is higher processing and memory costs.

A limitation is placed upon α, in a similar way. A higher α increases the number of grid-layers required. This results in more cycles to probe the grid and prepare the stream. Thus higher partition resolution also increases the processing and memory costs during pruning.

Understanding and quantifying these costs allows us to tailor the algorithm parameters for balance of pruning power, performance and memory costs.

In the worst case, no pruning occurs and X will contain all the pairs in the Cartesian product of A and B ($A \times B$). In the best case all the non-intersecting pairs are pruned and X will only contain pairs in the solution set, Q.

Quantifying X is not a trivial task. The following three sub-sections begin with a simplified scenario, where objects are assumed to be stationary and there is infinite grid resolution, then progressively constructs a version encompassing moving objects and a finite grid resolution.

5.1.1 Case 1: Stationary objects and infinite grid resolution, $v_{max} = 0$, $\gamma \to 0$.

Consider the process as defined in Algorithm 1. When an object $b \in B$ probes the grid and returns a negative result for partition $p \in P$, we effectively remove $\{a \times b | a \in p\}$ from X. Conversely, the set X is made up of all the tuples $\{a \times b | \forall a \in p$ where b returns a positive result for partition $p\}$. Thus X can be defined as follows.

$$X = \{a \times b | \forall b \in B, \forall a \in p, \forall p \in P : \exists \tilde{a} \in p, Intersects(\tilde{a}, b)\} \quad (2)$$

(Note: X is not constant because the equation involves the size of P, which depends upon the algorithm parameter α.)

In order to calculate $|X|$, we need to know how many pairs of a and b intersect each other. For this we introduce the term σ to define the selectivity of the query. σ takes a value between 0 and 1, and can be described as follows.

$$\sigma = \frac{|Q|}{|A \times B|} \quad (3)$$

It is also necessary to define the number of objects within each partition, β. β can be thought of as inversely proportional to α. A smaller α results in a larger β, and the associated performance and pruning changes. Thus, we have:

$$\beta = \left\lceil \frac{|A|}{\alpha} \right\rceil \quad (4)$$

Assuming that every pair in Q is in a different partition, then β object pairs will be added to X for every pair in Q, i.e., $|X| = \beta \times |Q|$.

$$|X| = \sigma\beta|A \times B| \tag{5}$$

Every pair in Q does not have to reside in a different partition, thus Equation (5) will contain duplicates. To avoid this, we refine Equation (5) to reflect the probability of b intersecting two objects in the same partition. Let σ be the probability that any two random objects a and b intersect. We require the probability, ϑ, that a random object b will intersect **at least one object** a from partition p. This is analogous to the probability that in β coin tosses, at least one toss will come up heads. Therefore,

$$\vartheta = 1 - (1 - \sigma)^{\beta} \tag{6}$$

Without a known σ, we cannot progress further. If we assume a uniform distribution in a finite region, the probability of a random point being within a given rectangle, placed randomly, is based on the proportion of the region the rectangle occupies. In this query, the area of the possible region is the entire domain, defined by width multiplied height, wh. As described earlier, the size of each object is xy. Equation (7) describes this probability Pr.

$$Pr = \frac{xy}{wh} \tag{7}$$

To extend this probability from a point within one object's extent to two object's extents overlapping, we consider the probability of a point lying within both object's extents. We know the likelihood of two independent events occurring is the product of their respective probabilities. Thus we compute σ as follows.

Without loss of generality, we assume that both the region and all objects' extents are square, i.e. $x = y$ and $w = h$ (this is done due to space constraints). Thus, the area of each object is x^2 and the area of the region is w^2.

$$\sigma = \frac{x^4}{w^4} \tag{8}$$

Combining Equation (6) and (8) defines our value $|X|$ as a function of known problem attributes (region size, object size) and partition size β, as shown in Equation (9).

$$|X| = \left\lfloor \left(1 - \left(\frac{w^4 - x^4}{w^4}\right)^{\beta}\right)|A \times B| \right\rfloor \tag{9}$$

(Note: $|X|$ is not static as it relies on β.)

5.1.2 Case 2: Stationary objects and finite grid resolution, $v_{max} = 0$, $\gamma > 0$

The analysis in the previous section excluded the effect the grid size plays on pruning. Equation (8) computes the probability that two objects intersect based on their extents. When the objects are placed on grids, our algorithm assumes that they intersect as long as they overlap the same cell, regardless of their precise extent. This effectively inflates the objects, rounding their extent up to fill the grid cells they partially occupy. This new extent can be determined.

To modify σ to account for the inflation, we introduce the term **effective size**, $\tilde{x}$. The effective size is the average length of a line x if placed randomly on the grid and increased to fill a number of grid cells of width γ. Probabilistically, $\tilde{x}$ is the expected value of x when represented by an integer number of grid cells, $\tilde{x} = E(x)$. It is described in Equation (10).

$$\tilde{x} = x \pmod{\gamma} + \left(1 + \left\lfloor \frac{x}{\gamma} \right\rfloor\right) \times \gamma \tag{10}$$

By replacing x^4 with $\tilde{x}^4$ in Equation (9) we have successfully integrated the effect of the granularity on the MLG-join algorithm's performance. See Equation (11).

$$|X| = \left\lfloor \left(1 - \left(\frac{w^4 - \tilde{x}^4}{w^4}\right)^{\beta}\right)|A \times B| \right\rfloor \tag{11}$$

5.1.3 Case 3: Moving objects and finite grid resolution, $v_{max} > 0$, $\gamma > 0$

Finally, we add velocity to the mix. The stationary rectangle used in the previous steps (defined by area x^2), must now be increased in size to bound the moving object over a period of time, ΔT. This was shown pictorially in Figure 4.

We will assume a uniform distribution of directions, and distribution of velocities between 0 and v_{max}, where v_{max} is the maximum velocity of any object. From this we can determine that the expected velocity is $\frac{v_{max}}{2}$. The expected area of the MBR, φ, is described in Equation (12).

$$\varphi = \left(x + \hat{x} \cdot \frac{v_{max}}{2}\Delta T\right) \times \left(y + \hat{y} \cdot \frac{v_{max}}{2}\Delta T\right) \tag{12}$$

where $\hat{x}$ and $\hat{y}$ denotes the unit vector in the $\mathbf{x}$ and $\mathbf{y}$ respectively. The vectors $\mathbf{x}$ and $\mathbf{y}$ are the coordinates of the space this problem exists within. (x, y) as defined earlier are the magnitudes of the length of each object's extent in this space. ΔT is the period of time the MBR is valid for.

To simplify the equation, we assume $x = y$ as in Section 5.1.1. As the velocity is uniformally distributed, it can be described as having any direction between 0 and π. Finally using some trigonometry, we arrive at Equation (13).

$$\varphi = x^2 + x \times v_{max} \times \Delta T \times \left(1 - \Delta T \times \frac{v_{max}}{8}\right) \tag{13}$$

The effective value of φ, $\tilde{\varphi}$, is computed by replacing x with $\tilde{x}$ from Equation (10). The resulting equation for $\tilde{\varphi}$ is omitted due to formatting constraints. $\tilde{\varphi}$ is placed

into Equation (11) in-place of $\tilde{x}^2$ to account for the area of moving objects. Finally α is substituted back in for β, following Equation (4).

$$|X| = \left\lfloor 1 - \left(\frac{w^4 - \tilde{\varphi}^2}{w^4} \right)^{\frac{|A|}{\alpha}} \right\rfloor |A \times B| \tag{14}$$

This equation correctly models both the initial join and the maintenance phases. In the maintenance phase only a portion of $|A \times B|$ is operated on, but this does not effect the choice of optimal parameters.

The definition of $|X|$ described in Equation (14) can be used to determine algorithm parameters to achieve maximum pruning. The accuracy of Equation (14) is validated experimentally in Section 6.2. The experimental results, coupled with this equation, allows us to choose the optimal parameters for our comparative analysis in Section 6.2.

6 Experiments

In this section a series of experiments are undertaken with three main goals: the first is to verify the accuracy of the analysis of Section 5 (see Section 6.2); the second is to perform a detailed experimental study of the key parameters, number of partitions (α) and grid size (γ), on the performance of the MLG-join algorithm (see Section 6.2); the third is to compare the MLG-join algorithm's performance against the current state-of-the-art algorithm (MTB-join) and a nested-loops join for GPUs (see Section 6.3). The experimental environment is first described along with the performance metrics.

6.1 Experimental Setup

The experiments were conducted using Microsoft Visual Studio 2010 development environment on 64 bit Windows 7. The proposed algorithm was compiled using CUDA Toolkit 4.0. It was selected over OpenCL due to the maturity of the documentation and development tools, however an OpenCL implementation is possible. To ensure validity of test results, operating system controlled context switching was disabled.

Initial join results were achieved by running the algorithm 10 times and reporting the median result. Maintenance results were achieved by *warming up* the index over 60 timestamps with updates, and averaging the cost of a single timestamp over the next 60. This process was repeated 10 times and the median result reported.

The host platform consisted of an Intel Core i7 950 3.06 Ghz CPU with 8 GB of RAM. The graphics card was an NVIDIA 1 GB GeForce GTX 460 SE, running at 1.3 Ghz.

Real Dataset. We follow Zhang et al. [26] and adopt two real-world trajectory datasets: a fleet of trucks and a fleet of school buses [7]. These two datasets consist of 276 and 145 trajectories, respectively, where each trajectory consist of the location points of a truck or a bus in a day. The location points are recorded at an interval of 30 seconds. Due to the limited size of the two datasets, following Zhang et al. [26] and a few other previous studies [6,8,13], we generate more objects based on the distribution and characteristics of these trajectories. Two sets of datasets are generated, i.e., the "Truck datasets" and the "Bus datasets", where the objects are generated based on the truck trajectories and the bus trajectories, respectively. We generate 100K objects for every dataset, where every object is generated as follows. We first randomly choose a trajectory from the corresponding real dataset for the object. We then randomly choose a location point of the trajectory as the object's starting position and give the object a size of 0.5% of the space (i.e., the default object size). We randomly choose one of the adjacent location points in the trajectory, and generate the current velocity for the object which will allow it to reach the chosen adjacent location point in 30 timestamps. When the object reaches the next location point in the trajectory, we repeat the adjacent location point choosing and velocity generating process, and an object update is issued.

Synthetic Dataset. Synthetic datasets were generated with a domain of 1000×1000 using the benchmark data generator (cf. [26]). Uniform and Guassian distributed datasets were used. In both datasets the speed was distributed uniformally between 0 and v_{max}. All objects had square extents. The maximum velocity ($v_{max} = 1$) and maximum update interval ($T_M = 60$) were fixed and adopted from Zhang et. al. [26]. Two datasets were initially setup at timestamp t_0. At each timestamp 1% of objects randomly updated their position, velocity and direction. Any object which did not update over an interval of T_M was removed from the datasets. The continuous query was first evaluated at t_0 and re-evaluated at every timestamp. The parameters used to vary the characteristics of the two input data sets A and B are summarised in Table 2.

The default **MLG-join algorithm** parameter values, unless otherwise specified, were $\alpha = 128$ and $\gamma = 1$. As mentioned in Section 4.3 we set ΔT to equal T_M which in our experiments was 60 timestamps. Table 3 displays the grid size for the range of parameter values used in our experiments. It is important to note even the largest grid (at $\alpha = 256$ and $\gamma = 0.5$) is only 64 MB in size.

Symbol	Description	Value
$\|A\|$, $\|B\|$	Cardinality	1K, 10K, 50K, **100K**, 500K
x, y	Object size (% of region size)	0.05%, **0.1%**, 0.2% 0.4% 0.8%
v_{max}	Object velocity (% of region size per second)	0.05%, **0.1%**, 0.2% 0.4% 0.8%
	Update rate (% per timestamp)	0.5%, **1%**, 2% 4% 8%
	Distribution of object position.	Gaussian, **Uniform**

Table 2 Parameters for synthetic datasets. Default values are in bold.

The **MTB-join algorithm** implementation and parameters used were obtained from Zhang et al. [26]. The I/O time was removed by implementing a RAM disk. The parameter ranges were chosen to be comparable with this study. It is important to note that the MTB-join is designed to work for disk-based data and therefore is not highly optimised for in memory continuous spatial joins. As future work, it would be interesting to explore ways of optimising the MTB-join for both in-memory data and GPUs.

A highly GPU optimized implementation of the **nested-loops join** was used in the study. This algorithm was designed with a similar execution structure as Bohm et. al. [3], with objects replacing points. The implementation was designed to obtain full occupancy on GPUs and 100% coalesced access patterns. This was confirmed using the CUDA Visual Profiler prior to testing.

A multi-core CPU version of the MLG-join was also implemented for comparison purposes. The implementation was similar to that of the GPU with some minor changes. The work of a partition was performed in small batches by a CPU thread, instead of a block of GPU threads. Alternative values were also used for α (number of partitions) and γ (grid size). Experiments were performed to select the optimal parameter values for the default dataset parameters (Table 3), the results of which are presented in Figure 8(b). The CPU implementation required significantly less memory for the same α value, as all the data structures, such as the streams, could be dynamically allocated. This allowed for much larger values for α to be tested.

	Grid cell size (γ)		
Number of partitions (α)	0.5	1	9
64	16 MB	8 MB	1 MB
128	32 MB	16 MB	2 MB
256	64 MB	32 MB	4 MB

Table 3 Grid size for varied parameters.

6.2 Validation of theoretical analysis

Equation (14) of Section 5.1, offers an insight into the pruning ability of the MLG-join algorithm. This ability is contingent upon selecting appropriate values for the number of partitions (α) and the size of grid cells (γ). To aid in selecting the best values, the effect these parameters have on the performance of the MLG-join algorithm is explored. This performance is evaluated in two ways, firstly the ability of the algorithm to prune the search space, secondly the cost (time) of running the algorithm.

The ability of the algorithm to prune the search space is quantified by measuring the percentage of pairs remaining in the Cartesian product of the two input datasets, defined by $\frac{|X|}{|A \times B|}$, where $|X|$ is the number of object pairs remaining after the pruning as defined in Section 5.1. This value is used on the y-axis of Figure 6(a) and 7(a). The cost of running the first three steps of the algorithm (build, probe, prepare) is examined first, followed by the cost of the entire algorithm. This analysis provides insight into the effect α and γ have upon the algorithm steps, in addition to allowing us to select the optimal parameter set.

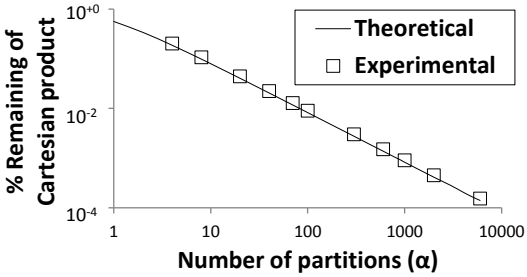
6.2.1 Effect of the number of partitions (α)

Figure 6(a) verifies the accuracy of Equation (14) as the experimental results align closely with those predicted by the equation for variations in α.

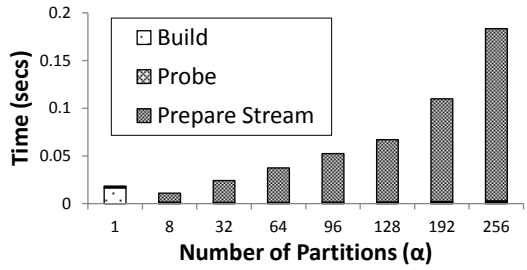
From Figure 6(a), we observe that α is positively proportional to the algorithm's pruning ability thus a higher number of partitions is preferred, as predicted by Equation (14). From Figure 6(b), we observe that above a low threshold (1 partition) α is positively proportional to the cost. Given these two behaviors, a compromise between pruning ability and cost is required. From Figure 6(c) we can see that 128 is the optimal value for α, and in the following experiments we will use this α value. At this value, approximately 99% of the Cartesian Product has been pruned.

The shape of Figure 6(a) is expected as a single partition is a very coarse grain approach which results in many false positives (non-intersecting pairs which are not pruned), whereas a partition per object is the finest possible grain resulting in very few false positives.

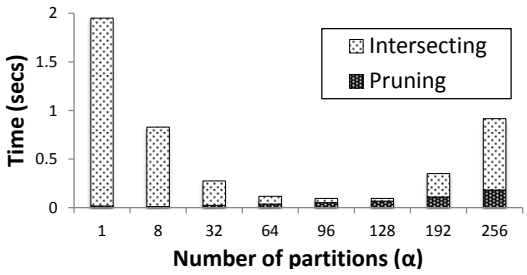
Figure 6(b) shows once there are a sufficient number of partitions to occupy the GPU hardware (8 or higher) the cost of building and probing the grid are fairly constant and minimal (under 1ms per step). The cost of stream preparation is the dominate step as a bit must be processed for each partition resulting in a linear scale.



(a) Pruning ability.

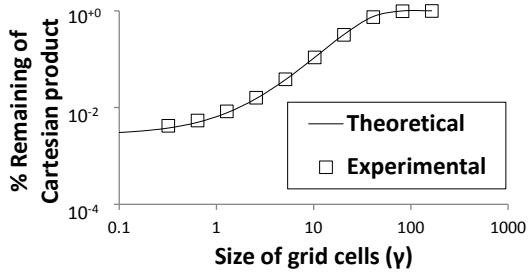


(b) Pruning time.

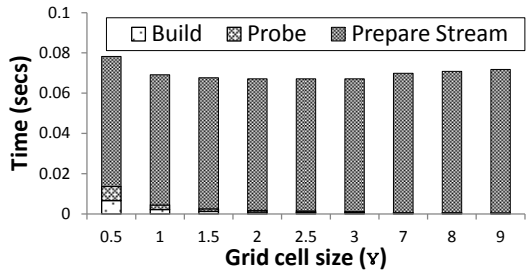


(c) Total time

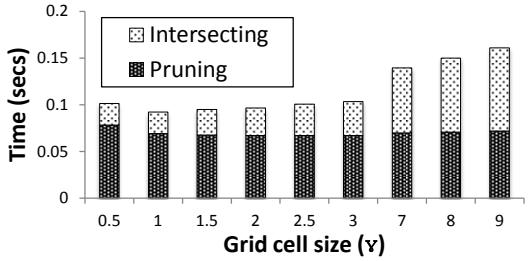
Fig. 6 Effect of α on MLG-join performance.



(a) Pruning ability



(b) Pruning time



(c) Total time

Fig. 7 Effect of γ on MLG-join performance.

One characteristic that may be counter-intuitive is the increase in intersecting time for large α in Figure 6(c). The increase in intersecting time is due to a decrease in parallel performance. A higher α results in fewer objects per partition, and consequently less parallel threads and operations.

6.2.2 Effect of the size of grid cells (γ)

Figure 7(a) verifies the accuracy of Equation (14) as the experimental results align closely with those predicted by the equation for variations in γ. From Figure 7(a), we observe that γ is negatively proportional to the pruning ability of the algorithm thus smaller grid cells are preferred. This is intuitive as smaller grid cells yield a finer grained join and less false positives. From Figure 7(b), we observed that above a low threshold of 1 the cost is approximately constant. By selecting the finest grained approach (smallest γ) above the threshold an optimal value is found (γ=1).

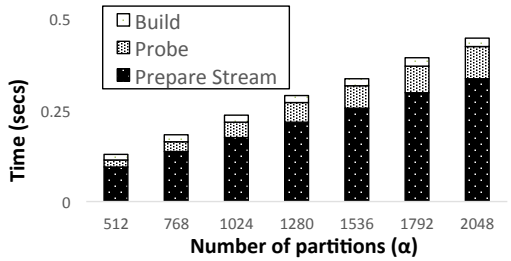
The optimal value is a trade-off between pruning and intersection times. A coarser grained grid (higher γ) is faster to build and probe, but yields more false positives, thus more objects are added to the stream for the intersection step. Whereas, a finer grained grid is costly to build as each object overlaps more cells, but yields a smaller stream. The exception to this relationship occurs for values below 1. As the objects used in this experiment had size 1, the benefit of increasing grid resolution beyond 1 has little impact on the stream size but increased the cost of grid construction.

Figure 7(c) shows the results for the total time, including both pruning time and intersecting time. The results reinforce our earlier expectation, namely the smallest value of γ that is higher than object size (1) yields the lowest total time. Therefore the optimal value of γ is 1, which is used for the remaining experiments.

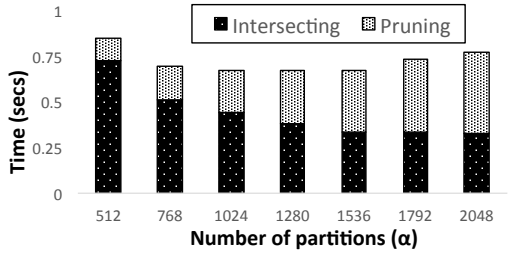
6.2.3 Parameter tuning for the CPU-based MLG-join

The optimized multi-threaded MLG-join algorithm on the multi-core CPU required different parameters to the GPU version ($\alpha = 1536, \gamma = 1$). It is expected that the build and probe steps will be much faster on the CPU

compared to the GPU as they involve non-coalesced memory access, a task that the CPU accomplishes easily. This can be seen in Figure 8(a) where the build step is almost negligible.



(a) Pruning time.



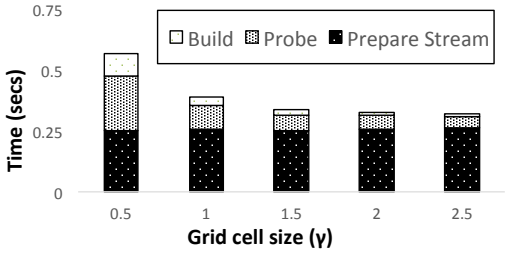
(b) Total time

Fig. 8 Effect of α and γ on CPU-based MLG-join performance.

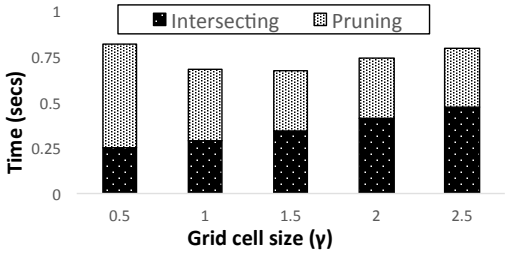
It should be noted that the varying α values (number of partitions) CPU graphs included larger α values than the corresponding GPU graphs. This is due to three factors. The first is memory limitations, as the CPU has much more memory than the GPU it can handle a larger grid, longer bit-strings for the probe and more streams for the additional partitions (all scale linearly with α). The second is that the GPU requires a large numbers of threads per block to achieve full occupancy and hide memory latency, so a large number of small partitions is not suitable. The final factor is that the CPU performs optimally with many small partitions, rather than the GPU which prefers fewer, larger partitions.

The CPU prefers small partitions as this reduces the workload of the intersecting step. More partitions greatly increased the ability of the grid to prune the search space, as can be seen in Figure 6(a). Unlike the GPU, the CPU cannot efficiently spread the workload of the intersection over hundreds of threads thus, a smaller workload is highly beneficial.

The cost of this finer pruning is expensive stream preparation. As each partition requires a separate stream, and the input to the stream preparation process is proportional to the number of partitions (bit-string length), the time required to prepare the stream scales linearly with the number of partitions. This is shown by Figure 8(b) where the GPU finds optimal performance at around 1500 partitions. At this stage, the increased cost of preparing the stream begins to counteract the benefit of a smaller intersecting step.



(a) Pruning time.



(b) Total time

Fig. 9 Effect of γ on CPU-based MLG-join performance.

The optimal value of γ was found to be 1.5. As γ increases in value the cost of the build and probe section becomes smaller, due to the reduction in cells each object overlaps. Conversely, as γ increases in value so does the cost of processing the intersections, as the coarser grid gives more false positives. Thus a trade-off must be found, which occurs at approximately $\gamma = 1.5$.

6.3 Performance comparison

The optimized MLG-join algorithm's performance is now compared to the current state-of-the-art MTB-join algorithm and a GPU nested-loops join. There are three factors to consider when comparing the algorithms. The first is the *setup time*, which includes allocating required memory, transferring data to the GPU and building data structures. The second is the *initial join time*, which is the first join once the data structures are prepared. As future timestamps only require the solution be maintained, this is likely the only time at which the entire solution is computed at once. Finally the *maintenance time* (per timestamp) is measured. This is the amount of time it takes, per timestamp, to maintain the validity of the data structures and update the moving join solution, this includes transferring data to the GPU. All experiments use the default parameters as defined in Table 2 except where stated otherwise.

6.3.1 Varying distribution of the dataset

Figure 10 presents the three stages of the join separately, note the log-scale y axis. The MLG-join algorithm significantly outperforms the MTB-join algorithm in all stages and for both data distributions. The setup time of the MLG-join algorithm outperforms the MTB-join algorithm by over three orders of magnitude. The MLG-join algorithm significantly outperforms the MTB-join algorithm for both the initial join and join maintenance by up to 2 orders of magnitude. The reason for this is the MLG-join is able to take advantage of the massive processing power and memory bandwidth of the GPU while pruning many of the comparisons using the multi-layered grid. The MLG-join algorithm also achieves 2 orders of magnitude improvement in performance over the GPU nested-loops join due to the MLG-join's ability to prune the number of comparisons.

In the setup stage the distribution has minimal impact. The GPU based nested-loops join algorithm and MLG-join algorithm have similar costs as they both require basic array allocation and data to be transferred to the GPU. The MLG-join algorithm is slightly more expensive as the grid must also be allocated. The MTB-join algorithms setup time is significantly higher as building the TPR-trees within it are very costly. This exemplifies the difference between light-weight grid structures and complex tree structures, an important drawback of the MTB-join algorithm as it describes a severe latency before the initial result can be computed. This cost for setup renders the MTB-join algorithm unable to provide real-time results to ad-hoc queries.

Second we observe that the MTB-join algorithm's initial join stage suffers a higher performance penalty than the maintenance stage. This is likely due to the lower selectivity resulting in more joins initially.

Third we observe the performance of the MLG-join algorithm for the Gaussian dataset is better than the uniform dataset for the initial join. As the objects are grouped in more dense clusters, the grid structure becomes more efficient at representing them, resulting in less work during the stream preparation step.

6.3.2 Real datasets

Figure 12 shows the results comparing the performance of the different algorithms for the real dataset. The y-axis is again in log scale. We only show the results for maintenance because we found the results for setup time and the initial join varied very little across datasets. The results again show that the MLG-join algorithm significantly outperforms both the GPU Nested-Loops join and the MTB-join. The reason for this is the same as for the synthetic datasets, namely, MLG-join can prune the number of comparisons while taking advantage of the massive processing power and memory bandwidth of the GPU.

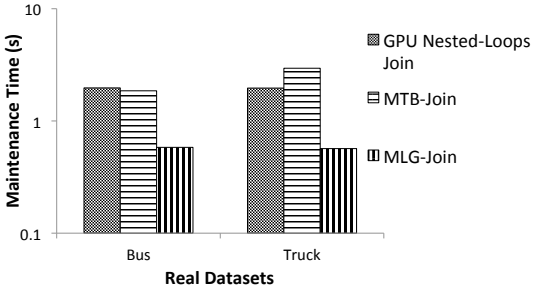


Fig. 12 Maintenance time for real data.

6.3.3 Multi-core CPU MLG-join versus GPU MLG-join

Figure 13 shows the results comparing the performance of a multi-core CPU implementation of MLG-join versus the GPU implementation. We only show the results for the join maintenance time since the setup time for the MLG-join is extremely low for both CPU and GPU implementations. Also the initial join and join maintenance results were very similar for the different MLG-join implementations. The experiment included both a single-threaded and a multi-threaded implementation of MLG-join. Different parameter values (α and γ) were used for the CPU and GPU implementations, based upon the optimal values found in Section 6.2.

The results show the GPU-based MLG-join algorithm outperforms the CPU implementations for both small and large datasets. The performance gap grows with increased cardinality. The GPU implementation outperforms the CPU implementations by 30 times at 500000 objects.

From Figure 13 we see the CPU version benefits from an increased number of threads. Whilst this may suggest a compute-bound algorithm, this is not necessarily the case as an increase in threads also provides access to additional core-specific cache (L1 and L2).

The GPU has approximately 4 times the memory bandwidth of the CPU which accounts for a portion of the performance difference. The remainder is most likely due to the way MLG exploits the broadcast capability of the GPU architecture. As mentioned in Section 4.2, when the GPU performs the intersection step on a partition the stream of objects is loaded into shared memory and broadcast to each thread. This results in a reduction in the number of global memory operations required. It is possible the increased cache available on the multi-threaded CPU provides a similar functionality but on a much smaller scale.

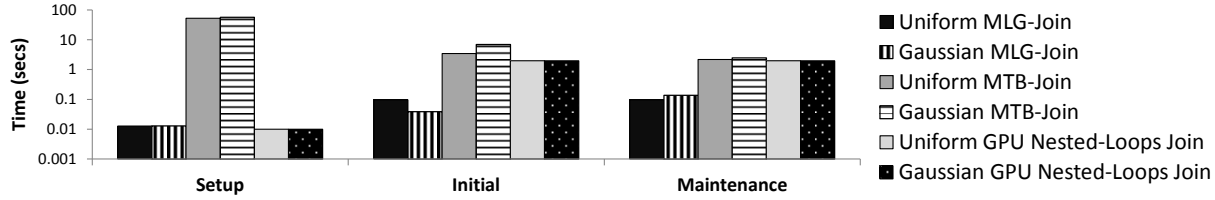


Fig. 10 Cost when varying dataset distribution.

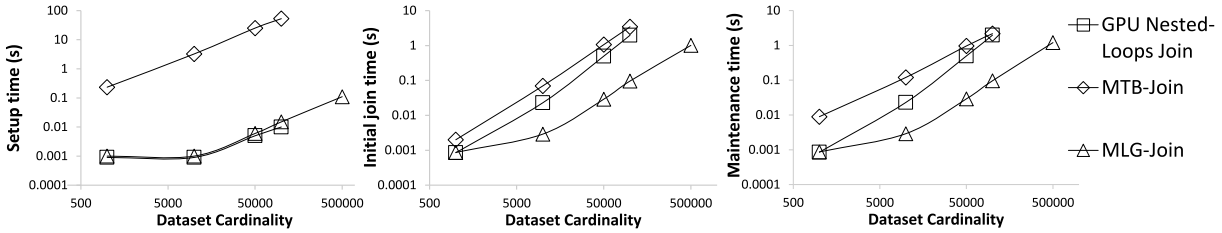


Fig. 11 Cost when varying dataset cardinality.

There is also a significant difference in raw compute-power between the CPU and GPU used. Namely, the total number of operations per second is much higher on the GPU than the CPU. It should also be noted that SIMD is not used by the CPU version of the MLG-join algorithm.

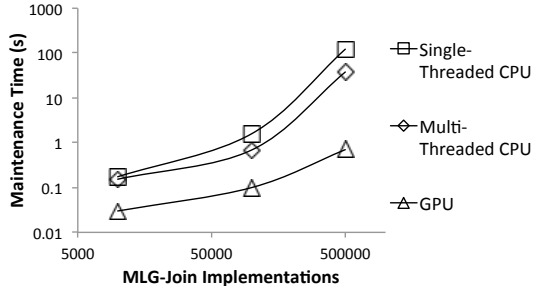


Fig. 13 Maintenance time for cardinality on CPU and GPU.

The purpose of this experiment was not to put the CPU against the GPU, but rather to compare the impact the different architectures have upon the algorithm. The MLG-join algorithm has several design choices that are counter-intuitive and inefficient for CPU-based processing, such as preparing the streams, pre-allocating large arrays rather than using dynamic structures, and grouping objects randomly without consideration for spatial location. All of these decisions likely contributed to the poor-scaling exhibited by the CPU implementation. However, on the GPU these inefficient decisions allow the GPU to operate at close to full capacity.

6.3.4 Effect of cardinality of the dataset

Figure 11 presents the algorithm running times as the data cardinality is varied. No results are shown for the MTB-join and the GPU nested-loops join for 500k objects due to their unreasonably long execution times at 500k objects. Keeping in mind that the x-axis is in log to base 10 scale, the results clearly show that MLG-join is much more scalable than its counterparts. For example it takes MTB-join less time to process the initial join and maintenance for 500k objects than it takes its counterparts to perform the same operations for 100k objects.

It is important to note the maintenance results include data transfer from the CPU to the GPU. Although this transfer time can be a bottleneck for some applications, for our experiments we found it only occupied a small portion of the overall maintenance time.

Figure 11 also shows for smaller datasets all algorithms perform similarly for the initial join. This is due to the minimal impact pruning has on small search spaces, and the under utilization of the GPU due to insufficient load to achieve full throughput. As cardinality increases, pruning becomes more cost-effective yielding performance gains for the MLG-join algorithm relative to the other algorithms.

6.3.5 Varying selectivity of the dataset

As described in Section 6.1 the selectivity is controlled by varying the size of the objects, whilst keeping the domain they exist within constant. Increasing the object size increased the likelihood of intersection. The setup time is not affected by selectivity and is thus omitted.

From the initial join results in Figure 14 we observe the GPU nested-loops join is not affected by varying selectivity because it processes all the search space regardless. The MLG-join algorithm is negatively effected by lower selectivity because its performance benefit derives from pruning the search space, lower selectivity results in less space that can be pruned. It is also likely, with larger objects, a different value for γ will yield superior performance. From the maintenance results in Figure 14 we observe an improvement in the MTB-join algorithm, compared to the initial join, due to its efficient updating. As the MLG-join algorithm does not

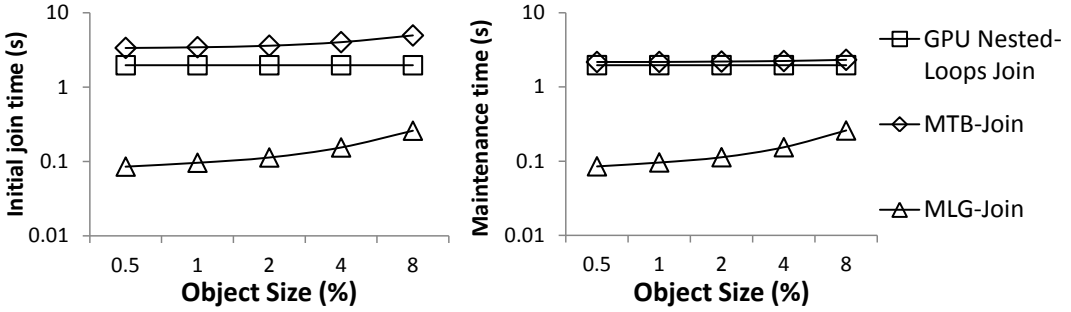


Fig. 14 Cost when varying object size.

move objects into different partitions based upon updating behavior, it cannot benefit in the same way.

6.3.6 Varying velocity of the dataset

From Figure 15 we observe that the MTB-join and MLG-join algorithms are similarly impacted by the variation of object velocity. This is expected as both algorithms model the locations of moving objects using expanding MBRs. The increased velocity also impacts upon the CI-join selectivity, as higher velocity objects transition in and out of intersecting locations more frequently. The GPU nested-loops join algorithm is unaffected by velocity variation as the workload is still fixed.

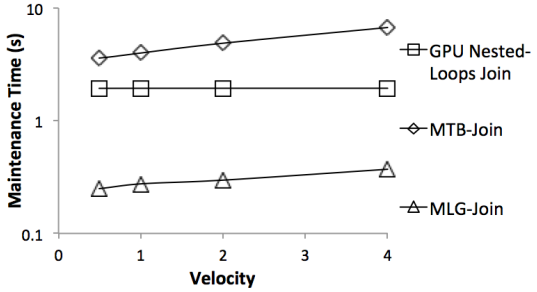


Fig. 15 Cost when varying object velocity.

6.3.7 Varying update frequency

From Figure 16 we observe that the MLG-join algorithm and GPU nested-loops join algorithm are largely unaffected by the varying update rate. This is expected for the GPU nested-loops join algorithm as the workload is largely fixed and no structure maintenance is required. The MLG-join algorithm performance reinforced the claim that the grid structure has a small maintenance cost. In contrast, the MTB-join algorithm is significantly impacted by higher update rates. This is due to the high maintenance cost of the TPR-tree as each update for the MTB-join may propagate to multiple nodes in the tree.

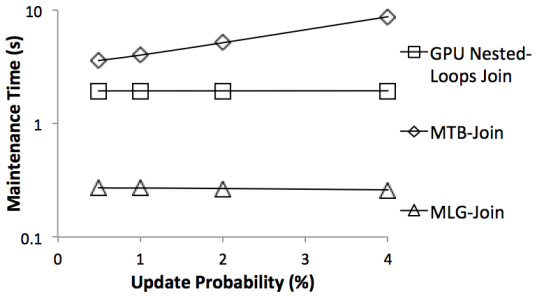


Fig. 16 Cost when varying update rate.

6.3.8 Summary of results

The MLG-join algorithm outperforms the current state-of-the-art (MTB-join) algorithm in all scenarios and by several orders of magnitude in most cases. The MLG-join algorithm's performance is not solely due to superior hardware as it also significantly outperforms the GPU nested-loops join algorithm.

7 Discussion

The approach of the MLG-join is unique in that it opts for reduced pruning power in return for inexpensive maintenance and GPU friendly execution of the unpruned pairs. This reduced pruning power is due to treating an entire partition as a single object in the pruning stage. This results in a best-case scenario where each actual intersection (object-object pair) results in a partition-object pair still to be processed. The silver lining to this outcome is that the partition can be processed in parallel and the memory requirements are coalesced. These two characteristics can result in the partition-object pair being processed in the same time as an object-object pair due to the nature of GPU hardware.

The grid approach, using inflated MBRs to account for motion, is very simple. Whilst it can be costly in the way it exaggerates object sizes, it is inexpensive to build and maintain. In addition, the underlying arrays are unsorted which removes the cost of keeping track of dynamic objects that many spatial structures incur. The high performance available on a GPU with simple

algorithms, such as the nested-loops join, sets a high standard for any pruning-based algorithm. That is, the pruning must incur a smaller cost per pair than the join algorithm would require to test it. The results in this paper suggest that the cost of maintaining more complex structures may not be cost-effective on the GPU.

The preference for simple algorithms on the GPU has encouraged brute-force, divide-and-conquer approaches to be highly successful. The MLG-join can be considered a divide-and-conquer hash approach. The divide is the partitions. Set A is divided into partitions, each of which is processed separately. The hash is the grid lookup, which provides a value relating the spatial location of an object to the memory location of a set of objects. The GPU is then highly suitable to process this set of objects in parallel.

A further benefit of the MLG-join is the lack of reliance upon sorted or clustered data. The random allocation of objects to partitions results in a natural distribution of any skew, resulting in intrinsic load balancing. Other structures, such as the R-tree, require expensive calculations during insertion to balance load and ensure optimal insertion positions for new objects. In highly dynamic, potentially skewed and high-response scenarios these added costs impede fast solutions and incur waste as objects may be updated or removed before the intended benefits of their organisation come into fruition.

8 Conclusion

We have proposed a GPU based algorithm called the MLG-join for the CI-join query. This algorithm utilizes contiguous access patterns to produce a grid representation of the underlying data, achieving both work and data load balancing across tens-of-thousands of execution threads on GPUs. This grid representation is then exploited to prune the number of comparisons, drastically reducing the computational workload. A thorough theoretical analysis of the algorithm behavior was undertaken, comprehensively supported by an empirical study with accuracy above 99.9%. This analysis assists the selection of optimal algorithm parameters.

Extensive experimental results show that the MLG-join algorithm outperforms the start-of-the-art CPU-based approach (MTB-join algorithm), and a naive GPU based algorithm (nested-loops join), by up to two orders of magnitude for executing the initial join and join maintenance. It also outperforms the MTB-join algorithm in setup time by four orders of magnitude. For future work we will apply our ideas to other types of queries, e.g. range and kNN queries.

Acknowledgments

This work is partly supported by the Australian Research Council's Discovery funding scheme (project number DP130104587). Rui Zhang is supported by the Australian Research Council's Future Fellow funding scheme (project number FT120100832).

References

1. M. E. Ali, E. Tanin, R. Zhang, and L. Kulik. A motion-aware approach for efficient evaluation of continuous queries on 3d object databases. *The VLDB Journal*, 19(5):603–632, 2010.
2. N. Bandi, C. Sun, A. El Abbadi, and D. Agrawal. Hardware acceleration in commercial databases: A case study of spatial operations. In *VLDB*, pages 1021–1032, 2004.
3. C. Böhm, R. Noll, C. Plant, B. Wackersreuther, and A. Zherdin. Data mining using graphics processing units. *Transactions on Large-Scale Data- and Knowledge-Centered Systems I*, 1:63–90, 2009.
4. J.-S. Boulanger, J. Kienzle, and C. Verbrugge. Comparing interest management algorithms for massively multiplayer games. In *NETGAMES*, page 6, 2006.
5. A. Corral, M. Torres, M. Vassilakopoulos, and Y. Manolopoulos. Predictive join processing between regions and moving objects. In *ADBIS*, pages 46–61, 2008.
6. H. Ding, G. Trajcevski, and P. Scheuermann. Omcat: optimal maintenance of continuous queries' answers for trajectories. In *SIGMOD*, pages 748–750, 2006.
7. E. Frentzos, K. Gratsias, N. Pelekis, and Y. Theodoridis. Nearest neighbor search on moving object trajectories. In *SSTD*, pages 328–345, 2005.
8. R. H. Güting, T. Behr, and J. Xu. Efficient k-nearest neighbor search on moving object trajectories. *VLDB Journal*, 19(5):687–714, 2010.
9. A. Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD*, pages 47–57, 1984.
10. B. He, K. Yang, R. Fang, M. Lu, N. K. Govindaraju, Q. Luo, and P. V. Sander. Relational joins on graphics processors. In *SIGMOD*, pages 511–524, 2008.
11. J. Huang, Z. Wen, J. Qi, R. Zhang, J. Chen, and Z. He. Top-k most influential locations selection. In *CIKM*, pages 2377–2380, 2011.
12. C. S. Jensen, D. Lin, and B. C. Ooi. Query and update efficient b^+-tree based indexing of moving objects. In *VLDB*, pages 768–779, 2004.
13. H. Jeung, M. L. Yiu, X. Zhou, C. S. Jensen, and H. T. Shen. Discovery of convoys in trajectory databases. *Proc. VLDB Endow.*, 1(1):1068–1080, 2008.
14. C. Kim, J. Chhugani, N. Satish, E. Sedlar, A. D. Nguyen, T. Kaldewey, V. W. Lee, S. A. Brandt, and P. Dubey. Fast: fast architecture sensitive tree search on modern cpus and gpus. In *SIGMOD*, pages 339–350, 2010.
15. K. L. Morse, L. Bic, and M. B. Dillencourt. Interest management in large-scale virtual environments. *Presence: Teleoperators and Virtual Environments*, 9(1):52–68, 2000.
16. J. Qi, R. Zhang, L. Kulik, D. Lin, and Y. Xue. The mindist location selection query. In *ICDE*, pages 366–377, 2012.
17. N. Roussopoulos, S. Kelley, and F. Vincent. Nearest neighbor queries. In *SIGMOD*, pages 71–79, 1995.

18. M. A. V. Salles, T. Cao, B. Sowell, A. J. Demers, J. Gehrke, C. Koch, and W. M. White. An evaluation of checkpoint recovery for massively multiplayer online games. *PVLDB*, 2(1):1258–1269, 2009.
19. S. Saltenis, C. S. Jensen, S. T. Leutenegger, and M. A. Lopez. Indexing the positions of continuously moving objects. In *SIGMOD*, pages 331–342. ACM, 2000.
20. D. Sidlauskas, S. Saltenis, and C. S. Jensen. Parallel main-memory indexing for moving-object query and update workloads. In *SIGMOD*, pages 37–48, 2012.
21. Y. Tao, D. Papadias, and J. Sun. The tpr*-tree: An optimized spatio-temporal access method for predictive queries. In *VLDB*, pages 790–801, 2003.
22. O. Wolfson, B. Xu, S. Chamberlain, and L. Jiang. Moving objects databases: Issues and solutions. In *SSDBM*, pages 111–122, 1998.
23. J. Zhang and S. You. Speeding up large-scale point-in-polygon test based spatial join on GPUs. In *Proceedings of the 1st ACM SIGSPATIAL International Workshop on Analytics for Big Geospatial Data*, pages 23–32, 2012.
24. J. Zhang, S. You, and L. Gruenwald. Parallel online spatial and temporal aggregations on multi-core CPUs and many-core GPUs. *Information Systems*, 2014.
25. R. Zhang, D. Lin, K. Ramamohanarao, and E. Bertino. Continuous intersection joins over moving objects. In *ICDE*, pages 863–872, 2008.
26. R. Zhang, J. Qi, D. Lin, W. Wang, and R. C.-W. Wong. A highly optimized algorithm for continuous intersection join queries over moving objects. *VLDB J.*, 21(4):561–586, 2012.