2014 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing

GPU-based Computing of Repeated Range Queries over Moving Objects

Claudio Silvestri, Francesco Lettich, and Salvatore Orlando
Università Ca' Foscari Venezia, Italy
Email: {lettich,orlando,silvestri}@dais.unive.it

Christian S. Jensen
Aarhus University, Denmark
Email: csj@cs.au.dk

***Abstract*—In this paper we investigate the use of GPUs to solve a data-intensive problem that involves huge amounts of moving objects. The scenario which we focus on regards objects that continuously move in a 2D space, where a large percentage of them also issues range queries. The processing of these queries entails a large quantity of objects falling into the range queries to be returned. In order to solve this problem by maintaining a suitable throughput, we partition the time into ticks, and defer the parallel processing of all the objects events (location updates and range queries) occurring in a given tick to the next tick, thus slightly delaying the overall computation. We process in parallel all the events of each tick by adopting an hybrid approach, based on the combined use of CPU and GPU, and show the suitability of the method by discussing performance results.**

The exploitation of a GPU allow us to achieve a speedup of more than 20× on several datasets with respect to the best sequential algorithm solving the same problem. More importantly, we show that the adoption of new bitmap-based intermediate data structure we propose to avoid memory access contention entails a 10× speedup with respect to naive GPU based solutions.

I. Introduction

An increasing number of applications rely on the processing of massive spatial and spatio-temporal workloads. Specifically, we consider applications in settings where moving object data are continuously produced and need to be processed rapidly, such as mobile phone infrastructures, Massively Multiplayer Online Games, and behavioral simulations where the agents may affect the behaviors of other agents within a given range. In these applications, very large populations of continuously moving objects frequently update their positions and issue queries for other objects within their range. The resulting massive workloads pose new challenges to data management techniques.

Figure 1 shows an instance of this setting with three objects. The object positions are represented as circles with the object ids inside. Updates are shown as previous (in gray) and new positions connected by arrows labeled u_1, u_2, u_3, and queries are shown as rectangles, labeled q_1, q_2, q_3. The result of query q_3 when executed after u_2 is $\{o_2, o_3\}$.

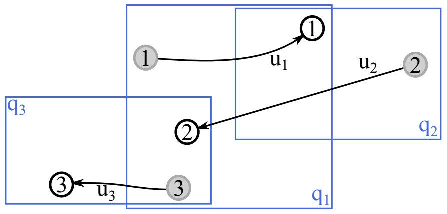

Fig. 1: Moving objects, queries, and position updates.

In order to contend with the targeted workloads in a scalable manner, we partition time into intervals (called *ticks*), assign the location updates and range queries to the ticks in which they occur, and process updates and queries in the resulting batches such that we report query results at the ends of ticks. This approach has the effect of replacing the processing of a large number of single, asynchronous object events (location updates and queries) with the repeated processing of spatial joins between the object positions as of the end of a tick and the range queries that have arrived during the tick. Our solution trades slightly delayed processing of queries for increased throughput, and special care is needed to ensure that this delay is acceptable.

To achieve high performance we exploit a platform encompassing an off-the-shelf general-purpose microprocessor (CPU) coupled with a *Graphics Processing Unit* (GPU) that features thousands of processing cores. To benefit from the computational power of a GPU, peculiarities and limitations of its architecture must be taken into account. Specifically, individual GPU cores are slower than those of a typical CPU and have memory access limitations that may cause contention. Effective query processing techniques must contend with these limitations and must ensure coordination among the cores.

To speed-up the range query processing during each tick, in particular to prune objects during the filtering phase, we need to exploit a spatial index that stores the moving object positions. Since in this work we assume *uniform spatial distribution* of objects in the space, a *regular grid index* suffices. Grid indexes are reported to be particularly effective for update-intensive workloads [6] and they also allow an easy parallelization of the workload, since the problem space is naturally partitioned into independently solvable sub-problems. Due to its simplicity and the huge amount of updates per time tick, we simply re-build the grid-based index from scratch before processing the queries occurring in a tick. Each sub-problem deriving from the grid-based partitioning is in turn a data-parallel computation, where the same task, i.e., a query processing, must be repeated for all the range queries falling into the cell.

We exploit the GPU, by adopting an hybrid CPU/GPU approach, in all the phases of the computation: *(i)* to pre-process the data such that objects in the same index grid cell are stored consecutively to optimize memory access; *(ii)* to realize nested data parallelism to process queries, where the outer parallelism applies to *all the cells* and associated objects, whereas the inner parallelism applies to *all the queries* in each cell; *(iii)* to produce and transfer in parallel the list of object ids for each issued query.

The first step *(i)* is the easiest one and the least expensive to compute, since it requires to read the dataset, compute some statistics over the moving objects, and finally build the grid and sort the objects.

The second step *(ii)* may potentially suffer from load imbalance. However, since in this paper we limit our study to datasets in which moving objects and queries are spatially distributed in a uniform way, the amount of objects/queries associated with each cell of the index is approximatively the same. In addition, a range query may span in principle many grid cells. However, since in the application setting that we address it is common to have queries whose range areas are identical, we carefully dimension the grid cell size in order to upper limit the number of index cells (at most four) to inspect when processing a query.

Thus, in this paper we focus on the third step *(iii)*, which becomes the most challengeable one in this simplified setting. In order to avoid the use of blocking writes and to ensure high throughput, for this step we propose a novel bitwise intermediate data structure that supports an approach based on two-steps, during which an interlaced list of results can be concurrently written by using coalesced memory accesses without incurring in race conditions.

We run extensive experiments, comparing the proposed approach to the best known competitor on CPU and to a straightforward GPU-based alternative that exploits a simple synchronization mechanism to store the list of query results. We show that the use of our bitwise intermediate data structure entails a one order of magnitude performance gain of the proposed approach with respect to the naive GPU-based one. Further, whereas the naive approach has just a marginal advantage over the sequential competitor, the proposed approach consistently outperforms it by a 20× factor.
To the best of our knowledge this is the first work that exploits GPUs to efficiently solve repeated spatial range queries on continuously moving objects.

The paper is organized as follows: in Section II we describe the problem setting and state the problem addressed. In Section III we present our solutions, while in Section IV we show a deep experimental study. Finally, in Section V we cite the relevant related works, while in Section VI we draw some conclusions and possible future works.

II. Problem Setting and Statement

A. Problem Setting

We consider a set of points $O = \{o_1, \ldots, o_n\}$ moving in a two-dimensional Euclidean space $\mathbb{R}^2$, where the position of an object o_i is given by the function $pos_i : \mathbb{R}_{\geq 0} \rightarrow \mathbb{R}^2$ mapping time instants into spatial positions. These points model objects that issue position updates and range queries as they move.

Let $\mathcal{P}_i = \langle p_i^{t_0}, \ldots, p_i^{t_k}, \ldots \rangle$, $t_j < t_{j+1}$, be the time-ordered sequence position updates issued by o_i, where $p_i^{t_j} = pos_i(t_j)$ is a position update. A range query issued by object o_i at time t is denoted by $q_i^t = (x^a, x^b, y^a, y^b)$, where (x^a, y^a) and (x^b, y^b) are the lower left and upper right corners of a rectangle. Thus, $\mathcal{Q}_i = \langle q_i^{t_0}, \ldots, q_i^{t_k}, \ldots \rangle$, $t_j < t_{j+1}$, is the time-ordered sequence of queries issued by o_i.

Given the above, the most recently known position of o_i at time t, $t \geq t_0$, is denoted as $\hat{p}_i^t$ and defined as follows.

$$\hat{p}_i^t = p_i^{t_k} \in \mathcal{P}_i \text{ if } t_k < t \leq t_{k+1}$$

Similarly, the most recent query issued by o_i at time t, $t \geq t_0$, is $\hat{q}_i^t$.

$$\hat{q}_i^t = q_i^{t_k} \in \mathcal{Q}_i \text{ if } t_k < t \leq t_{k+1}$$

We assume that the processing of a query can be delayed to a certain extent in order to optimize the overall system throughput. We process queries using the most up-to-date information available.

Definition 1: [Result set of a range query]
The result of query q_i^t when computed at time t', $t' \geq t \geq t_0$, is denoted by $res(q_i^t, t')$ and is defined as follows.

$$res(q_i^t, t') = \{o_j \in O \mid \hat{p}_j^{t'} \in_s q_i^t\},$$

where $\hat{p}_j^{t'} \in_s q_i^t$ denotes that $\hat{p}_j^{t'} = (x, y)$ belongs to the query rectangle q_i^t, i.e., $x^a \leq x \leq x^b$ and $y^a \leq y \leq y^b$.

Assuming that updates $u_1, \ldots, u_3$ in Figure 1 are the most recent ones before t', we have $res(q_3^t, t') = \{o_2, o_3\}$, which includes also object o_3 that issued the query.

B. Batch Processing

To obtain high throughput when facing massive workloads due to frequent updates and queries issued by very large populations of moving objects, we quantize time into time intervals, *ticks*, with the objective of processing updates and queries in batches on a per-tick basis. Assuming that the initial time is 0 and the tick duration is Δt, the k-th time tick τ_k is the time interval $[k \cdot \Delta t, \quad (k+1) \cdot \Delta t)$. Specifically, we aim to collect the updates and queries that arrive during a tick and process the updates and then the queries at the end of the tick. For simplicity and to be specific, we assume a setting where an unanswered query issued by an object becomes obsolete when the object issues a new query. Thus, if an object submits more than one update and one query during a time tick, only the most recent ones are processed. It is then straightforward to support scenarios where queries never become obsolete.

C. Query Semantics

The procedure for computing queries as described above ensures serializable query processing and implements the timeslice query semantics, where query results are consistent with respect to the database state at a given time, usually the time when the query processing begins. This is a popular choice in traditional databases and is commonly adopted in related work [7]–[9], [14]. We process updates and queries in batches, and thus query results are consistent with the database states at the boundaries of ticks when the processing of batches begin. When the computation finishes the results are returned.

D. Problem Statement

With the preceding definitions in place, we can state the problem addressed.

Definition 2: [Repeated Range Query Problem]
Given (i) a set of n objects O, (ii) a partitioning of the

time domain into ticks $[\tau_k]_{k\in\mathbb{N}}$ of duration Δt, and (iii) a sequence of pairs $[(P_{\tau_k}, Q_{\tau_k})]_{k\in\mathbb{N}}$, where P_{τ_k} is the up-to-date object positions at the end of τ_k, and Q_{τ_k} is the set of the last issued queries during τ_k, we define the repeated range query operator RRQ applied to $[(P_{\tau_k}, Q_{\tau_k})]_{k\in\mathbb{N}}$ to yield the sequence $[R_{\tau_k}]_{k\in\mathbb{N}}$, where

$$R_{\tau_k} = \{(q_i^{\tau_k},\ res(q_i^{\tau_k},\ k\cdot\Delta t))\ \mid\ q_i^{\tau_k} \neq \perp\ \wedge\ q_i^{\tau_k} \in Q_{\tau_k}\}.$$

In other words, R_{τ_k} is a set of pairs of a query and the list of its result.

Before moving to the description of the proposed methods, we rapidly cover background information on the type of hardware that we use to execute them.

E. Graphics Processing Units

GPUs are based on massively parallel computing architectures that feature thousands of *cores* grouped in *streaming multiprocessors* (SMs) and high-bandwidth RAM which can be used successfully for general purpose computing. The large number of available cores offers a potential for substantial performance gains when compared to the performance of traditional CPUs.

However, due to the architecture of these devices, it is non-trivial to exploit their computational power. Specifically, each GPU processing core is slower than a typical CPU and has limitations on its access to device memory, resulting in potential contentions unless specific conditions are satisfied [16]; the many cores need to coordinate their actions, which is non-trivial with large numbers of cores.
Special algorithms are needed, that are designed with the architecture of GPU in mind, which is not always possible or favorable [19]. in general, it is not possible to use an existing algorithm, even if conceived for a multi-core CPU.

To effectively exploit the computational power of a GPU, memory accesses should have high spatial and temporal locality to ensure that several cores can benefit of the same memory transfers (coalescing) and to avoid serialization of potentially parallel memory accesses (with consequent performance degradation by orders of magnitude). Moreover, GPUs feature several types of memory ranging from registers to fast memory shared among groups of cores, to device global memory, which is slower but of significant size and is the contact point with the host CPU. To achieve high performance in the context of this quite complex memory hierarchy an explicit management of memory transfers between the different memories is most of the times fundamental.

Finally, workload partitioning is paramount when designing GPU algorithms. A GPU consists of an array of n_{SM} multi-threaded SMs, each with n_{core} cores, yielding a total number of $n_{SM} \cdot n_{core}$ cores. Each SM is able to run *blocks* of *threads*, with the threads in a block running concurrently on the cores of an SM. Since a block typically has many more threads than an SM has cores, only a subset of the threads can run in parallel at a given time instant. Such subsets of threads, called *warps*, consist of sz_{warp} synchronous, *data parallel threads* and are executed according to an SIMD paradigm [12], [16]. Due to the SIMD model, it is important to avoid branching inside the same warp to assure optimal use of resources. At the warp level, no synchronization mechanisms are needed to guarantee data dependencies among threads.

III. Repeated Range Queries

We first cover design considerations underlying the join techniques. Then we cover indexing and present the GPU-based algorithms.
When processing a repeated batch of range queries, the same procedure is repeated for each tick. Thus, for the sake of readability, hereinafter we omit the subscript that indicates the tick, and denote by P, Q, and R, respectively, the up-to-date object positions, the non-obsolete queries, and the result set associated with a generic tick.

A. Design Considerations

A brute-force approach for computing a repeated set of range queries during a tick entails $O(|P|\cdot|Q|)$ containment checks. By introducing a proper spatial indexing it is possible to prune pairs of query ranges and object locations that cannot join. When choosing or designing an appropriate index, its pruning power is but one consideration. The cost of maintaining the index is another aspect to take into consideration. For example, grid indexes are reported to be particularly effective for update-intensive workloads [6].

Another consideration is the number of cores and memory hierarchy provided by the underlying computing platform: an index may perform differently on different platforms, so a specific one must be considered when choosing the best option. With massively parallel platforms such as GPUs, the regularity of grid indexes is attractive as it enables efficient parallel index update and querying. The location in the index of the data related to a given region of space is known a priori. For all these reasons, we thus adopt a *regular grid index*.

In the following we discuss this index, and then how queries are processed and submitted on a per grid cell basis. Since we assume that queries are homogeneous in size, we choose a grid cell size such that a query intersects at most *four cells*, thus limiting the search space.

B. Grid-Based Partitioning and Indexing

Let $\mathcal{G} = (x_a^{\mathcal{G}}, y_a^{\mathcal{G}}, x_b^{\mathcal{G}}, y_b^{\mathcal{G}})$ be the minimum bounding rectangle that contain all objects and query locations during the tick. We cover $\mathcal{G}$ by a regular grid and thus partition $\mathcal{G}$ into cells [11]. Then we enumerate the cells using a space-filling curve and map object locations and queries to cells as detailed next.

1) MBR partitioning into a regular grid: We partition $\mathcal{G}$ according to a grid $\mathcal{C}$ of $N\cdot M$ cells of width W and height H such that cell c_{ij} covers the following region:

$$(x_a^{\mathcal{G}} + i\cdot W,\quad x_a^{\mathcal{G}} + (i+1)\cdot W,\quad y_a^{\mathcal{G}} + j\cdot H,\quad y_a^{\mathcal{G}} + (j+1)\cdot H).$$

W and H are computed on the basis of the dataset features, in particular the area of the range queries occurring in a given tick. Therefore, given a set Q consisting of queries $q_i = \langle x_i^a, x_i^b, y_i^a, y_i^b\rangle$, the cell size $W\cdot H$ is computed as follows:

$$W = \max_{i=1,\ldots,|Q|} (x_i^b - x_i^a) \qquad H = \max_{j=1,\ldots,|Q|} (y_i^b - y_1^b)$$

Finally, to ensure that the grid covers $\mathcal{G}$, constants N and M are chosen so that $x_a^{\mathcal{G}} + N\cdot W \geq x_b^{\mathcal{G}}$ and $y_a^{\mathcal{G}} + M\cdot H \geq y_b^{\mathcal{G}}$ hold.

2) Mapping of objects and queries to cells: Next, we assume that the cells are enumerated according to the standard *z-curve* function such that the index of cell c_{ij} is $z(i, j)$. Function $f : P \rightarrow \mathcal{C}$ maps each object in P to the cell that contains the object. Function $g : Q \rightarrow \mathcal{C}$ maps each query in Q that is distinct from $\perp$ to the cell that contains its lower left corner, called its *primary cell*. We denote the range of function g by $\mathcal{C}_A$ and call it the set of *active cells*. Note that if g maps a query q to its primary cell c_{ij}, three other *neighboring cells* may intersect q: $c_{i(j+1)}$, $c_{(i+1)j}$, and $c_{(i+1)(j+1)}$. Thus, up to 4 cells need to be visited to compute a query. Figure 2 illustrates this scenario.

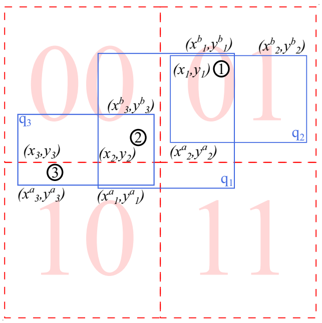

Fig. 2: Grid partitioning of the space

C. Query Processing Pipeline

Five main design issues underline our GPU-based solution: *(i)* we have to distribute the workload evenly among the GPUs' SMs; *(ii)* we have to avoid contention/serialization when accessing the GPU device memory and we must favor locality to fully exploit the complex GPU memory hierarchy; *(iii)* when possible, we have to apply data compression of the GPU output, which can be very large and much larger than the input for our problem; *(iv)* we have to avoid expensive synchronization among concurrent threads; and *(v)* we have to choose the unit of parallelization (either objects or queries, or partitions of queries).

The processing pipeline we propose is structured into five stages:

1) *index creation*;
2) *moving object and query indexing*;
3) *sorting*;
4) *filtering with bitmap encoding*;
5) *bitmap decoding and output*.

Each one of these stages takes advantage of a tight cooperation between the GPU and the CPU and exploits temporary ad-hoc *bit-wise data structures*, which encode the query result for a single tick, in order to address the design issues reported in *(ii)* and *(iv)*. Data are moved to GPU memory before stage 1 and wrote back to host memory during the last stage.

In order to show the benefits of using bitmap intermediate data structures, we also implement a *4-stage baseline pipeline*, where we replace the 4^{th} and 5^{th} stage above with the following one:

4) *filtering and synchronized output*.

In the following detailed pipeline description we will refer to the two pipelines as *baseline pipeline* and *optimized pipeline*. The common part, stages 1-3, will be described separately and indicated as *common pipeline*.

D. Common Pipeline Description

1) Index Creation phase: During the *index creation* phase the minimum rectangle $\mathcal{G}$ that bounds all objects and queries during the tick is first determined. Then, the grid index $\mathcal{C}$ that partitions $\mathcal{G}$ is created (see Section III-B1). This is done through simple reductions over P and Q. The overall complexity for both steps is $O(|Q| + |P|)$.

2) Moving Object and Query Indexing: This stage calculates the index of the cells to which the various queries and moving objects belong, given their coordinates. This is translated essentially into computing in parallel an integer value from a tuple of reals. This task has thus an overall complexity of $O(|Q| + |P|)$.

3) Sorting: During this phase we reorder separately P and Q accordingly to the indices of the cells computed in III-D2: the goal is to store objects and queries assigned to the same cell in contiguous memory locations so to improve *spatial locality* during the subsequent phases.

The sorting phase can be performed very efficiently on GPU by adopting a well established GPU-based sorting algorithm such as the radix-sort [5]. The complexity of this step is $O(d \cdot (|Q| + |P|))$, where d is the base used when sorting keys. Once we have reordered the vectors related to P and Q accordingly to the way described above, we also compute the start/stop indices needed to detect the lists of queries and moving objects belonging to each active cell.

E. Baseline Pipeline Description

4) Filtering and Synchronized Output: The 4^{th} and last stage of the baseline approach computes the intersection tests and writes out directly in the GPU's global memory. Each thread is in charge of a distinct query, and all threads concurrently enqueue the output to contiguous memory locations, as soon as a new object falling into a query is detected. The query results are thus produced by these threads as an un-ordered list of pairs $(query_{ID}, object_{ID})$.

This approach has two main drawbacks: (i) the need of using synchronizing mechanisms among threads (in each block) in order to flush out the results in global memory in a coherent manner, and (ii) the format of the output, which is not compressed at all, thus increasing the I/O costs. Since the implementation of the forth phase of the baseline pipeline is straightforward, we do not discuss it any more, although we discuss the results of some tests involving it in the experimental section.

F. Optimized Pipeline Description

4) Filtering with Bitmap Encoding: This phase actually computes the range queries against the object positions. Accordingly to the design issues cited in Section III-C, we have to exploit cleverly the GPU architecture in order to orchestrate efficiently the input/output going on between the CPU and the GPU. To this end we thus adopt a *multi-step* strategy, a thing which is rather common in the GPGPU field [5], [17], in which the phase described in this section represents the first step. The main idea employed during this phase is to store the results of the queries issued during a single tick to *bitmaps*. More specifically, for each cell $c \in \mathcal{C}$ and for all the queries whose primary cell is c, a 2D bitmap B^c is computed, where each row represents a single query assigned to c, and each column an object falling in either c or one of its (up to) three *neighboring cells* (see Section III-B2). The element b^c_{hk} is thus set to 1 if the h-th object is in the result of the k-th query.

Before computing the bitmaps, however, we must determine their size; this in turn depends on the convenience layout used to store the bitmaps, which is the *interlaced*, column-major one, as shown in Figure 3. Accordingly to this layout, the bits of each row are grouped in words $q_1, \ldots, q_p$, which yields a scattered access to the device memory when multiple GPU threads, each one in charge of computing a distinct query, create the bitmap. This, as it will be shown later, allows to coalesce the writes when bitmaps are stored inside the GPU's main global memory and to avoid the usage of synchronization mechanisms between threads.

The size of the bitmap for a given cell c_α is therefore $nq_\alpha \cdot np_\alpha$, where nq_α is the number of queries that map (by g) to c_α and np_α is the number of objects that can be in the range of a query in c_α. Thus, np_α is the number of objects in four cells: the (primary) cell $c_\alpha = c_{z(ij)}$ and its three neighboring cells $c_{z(i(j+1))}$, $c_{z((i+1)j)}$, and $c_{z((i+1)(j+1))}$, where z is the z-curve function.
The values nq_α and np_α can, for all k, be computed in parallel over the ordered vectors obtained during the sorting phase using a GPU-based parallel scan algorithm. At the end, along with the sizes of the bitmaps, we compute the starting addresses in global memory for the various bitmaps for each index cell as well.

After having determined the sizes of each bitmap we are finally ready to compute the spatial joins in parallel. Our GPU parallelization of the filtering and encoding phase is decomposed into two stages: *Interlaced bitmap generation* and *bitmap linearization*.

Interlaced bitmap generation. As shown in Listing 1, we have a distinct *block* of GPU threads for each active index cell c_α (line 5). Each thread in a block is then in charge of computing a distinct query assigned to c_α (line 6). Since there are usually many more threads/queries in a block than cores in an SM, only a subset of them can run in parallel at a given time. These subsets of threads are called *warps* and consist of sz_{warp} synchronous threads (32 data-parallel threads in our GPU setting). All the threads of a warp access the device memory in an optimized way: they read the same input data (object positions) (line 7) synchronously and access them consecutively (spatial locality). They first access the objects in $c_\alpha = c_{z(ij)}$, then the objects in $c_{z(i(j+1))}$, then those in $c_{z((i+1)j)}$, and finally those in $c_{z((i+1)(j+1))}$. Moreover, all threads write simultaneously words that are stored consecutively in memory, thus favoring coalescing of memory writes, which improves device memory bandwidth utilization (line 15).

In more detail, all threads read the same sequence of object positions and update locally a *32 bit-wide* register that contains the bitwise information about the presence/absence of 32 distinct objects in its own range query (line 11). When the threads in a warp have all completed the updating of their registers, all the threads store them simultaneously to the device memory at the right memory displacement (line 15). Then, they start computing the next words/registers, etc.

Listing 1: Filtering with bitmap encoding pseudocode

```
1  $numPoints \leftarrow 0$
2  $indexWord \leftarrow 0$
3  $wordBitmap \leftarrow 0$
4
5  foreach $c_\alpha \in \mathcal{C}_\alpha$ parallel GPU block do
6    foreach $q \in c_\alpha$ parallel GPU thread do
7      foreach $p \in c_\alpha$ do
8
9        $numPoints \leftarrow numPoints + 1$
10       if ($p \in q$)
11         $setBit(wordBitmap, p)$
12       endif
13
14       if ($numPoints$ mod $32 = 0$)
15         $writeBitmap(wordBitmap, indexWord, q)$
16         $wordBitmap \leftarrow 0$
17         $indexWord \leftarrow indexWord + 1$
18       endif
19
20     endfor
21   endfor
22 endfor
```

This process is also schematized in Figure 3, which shows the interlaced bitmap representation and a block of sz_{warp} columns that are collectively updated by a warp. Each column

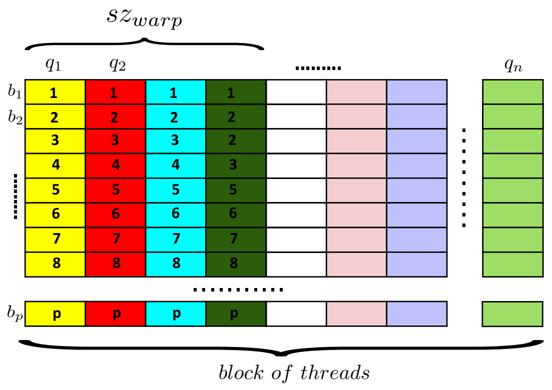

Fig. 3: Collective creation of a interlaced bitmap by a block of GPU threads.

contains the 32 bit-wide words associated with each query $q_1, \ldots, q_p$. The first words (the $b1$'s) associated with the various queries and computed by the warp's threads are stored simultaneously in memory at time 1. Then at time 2, the same holds for the second words (the $b2$'s), etc. The words updated simultaneously by the threads are stored consecutively due to the interlaced layout of the bitmap. This permits *coalesced writing* of data by the synchronous warp threads.
The complexity of the algorithm is:

$$\sum_{c \in \mathcal{C}_\alpha} |Q^c| \cdot |P^c|. \qquad (1)$$

To simplify the decoding of the information stored in the bitmaps, the interlaced bitmaps are then linearized. In the resulting layout (Figure 4), the bit-vectors associated with every queries have their words arranged consecutively in memory, which favors read coalescing during the decoding phase. This transformation can be done through a fairly simple GPU kernel.

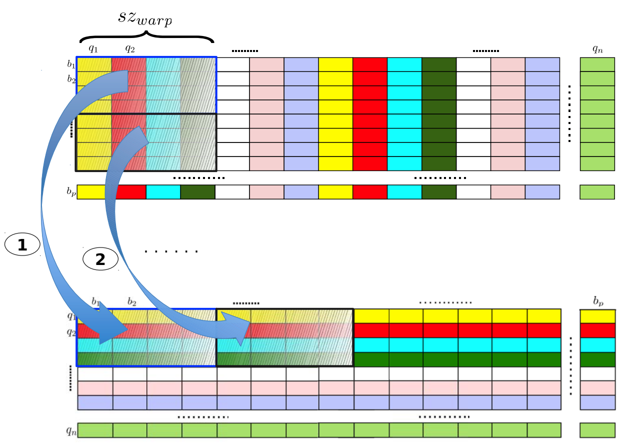

Fig. 4: Collective linearization of a group of columns by a warp of GPU threads.

5) Bitmap Decoding: The previous phase outputs one bitmap for each query, including both positive and negative occurrences of all object positions associated with a given index cell. From such a bitmap, the *bitmap decoding* phase has to generate a list of object identifiers corresponding to the positive occurrences.

It would be possible to transfer the linearised bitmaps from the GPU to the CPU and let the latter perform the decoding. However, the decoding is a computationally intensive task that can be massively parallelized. On the negative side, when carrying out the decoding using the GPU, we must transfer the results to the CPU. In sum, given the bandwidths of the buses through which CPUs and GPUs communicate, we choose to perform the decoding on the GPU.

We adopt a classical [15] solution to overlap the overhead of the communication between the CPU and the GPU with the computation on the GPU. In particular, we overlap the GPU's decoding of a bitmap chunk to produce a portion of the result, with the communication of another portion of the result, previously decoded by the GPU. A chunk is a subset of all the linearized bitmaps, produced by the previous pipeline stage for each index cell. To this end, we employ a *double buffer* technique. One buffer is used for GPU-based decoding of a chunk, while the other is used for transmitting already decoded results from the GPU to the CPU. Then, we exchange the roles of the buffers. This kind of overlapping is possible thanks to the hardware capabilities of GPUs, which allow simultaneous kernel execution and data transfer.

Each bitmap is decoded by a specific block of threads that are scheduled in warps over an SM. Queries are statically split among *warps*, the threads of which cooperate to decode the queries. First the threads load the bit vector words of the currently considered query into shared memory. Since the words are consecutive in memory in the linearized layout, the parallel reads are coalesced. Then, through a parallel double linear scan, the warp compute the numbers of the object identifiers (positive query results) to write to the device memory along with the right displacements.

The linear scan avoids bank conflicts, by exploiting the *broadcast* capability of shared memories, and it scales with respect to the size of the query bitmap. Once this information is computed, every thread inside a warp performs a collective write of the positive query result to global memory, exploiting write coalescing. This means that each thread in a warp writes consecutive words, each one containing the identifier of an object in the result set. The complexity of this phase can be expressed as:

$$|R| + \sum_{c \in \mathcal{C}_\alpha} |Q^c| \cdot |P^c|, \tag{2}$$

where $|R|$ is the part related to the I/O notification cost and the rest is related to the bitmaps' decoding costs.

IV. Experimental Evaluation

We present an experimental evaluation that aims to prove the effectiveness of using our intermediate bitmap data structure and to offer insight into effects of key parameters on the performance of the strategy we propose.

A. Experimental Setup

The experiments are conducted on a PC equipped with an Intel Core i3 550 CPU (at 3.2 GHz) with 4 GB of RAM and a NVIDIA GTX 560 GPU; the operating system used is Ubuntu 12.04, while the C/C++ compiler utilized is GCC 4.6.3 and the CUDA Toolkit version is 4.2.

We exploit a publicly available framework [1], [2] for both workload generation and testing. The framework comes with a number of sequential, CPU-based algorithms. Among these, *Synchronized Trasversal* is shown to be consistently the best [2], so we compare our methods against this one.

For the tests, we use *synthetic datasets* in which moving objects are distributed *uniformly* on the space. Workloads are created using the generator included in the framework, which is derived from the Brinkoff generator [18]. In all tests, we iterate the computing of a batch of object updates and range queries for 20 ticks. To model object movements, the framework thus generates 20 chunks of data for each dataset, one for each tick. Table I summarizes the main parameters used to generate the datasets. The framework uses a generic spatial distance unit u.

Spatial area (sa)	All tests occur in a square spatial area with size $500M\ u^2$ (side length $22,361\ u$).
Moving objs no. ($\|P\|$)	We vary the number of moving objects $\|P\|$ from $100k$ to $1.5M$.
Query rate (qr)	The percentage of objects that issue a range query during every tick, so that $\|Q\| = qr \cdot \|P\|$. Default value is 100%.
Query area (qa)	All query areas in our tests are squares of the same size. Default value is $40k\ u^2$ (side length $200\ u$).
Query location	qa's are centered in the position of objects issuing queries.

TABLE I: Data and workload generation parameters.

B. Experimental Results

In the following we report on some experimental results obtained by our GPU-based optimized software, denoted by $\mathsf{GPU_{Opt}}$, which exploits bitmaps during the filtering phase. We compare $\mathsf{GPU_{Opt}}$ with two baselines: the CPU-only implementation [2], denoted by Seq, and the GPU-based version, denoted by $\mathsf{GPU_{BLine}}$, which uses thread synchronization to correctly enqueue the list of results.

In the first batch of tests we study the behavior of $\mathsf{GPU_{Opt}}$ when we change the amount of moving objects, the query size, and the query rate. Figure 5 plots the execution time as a

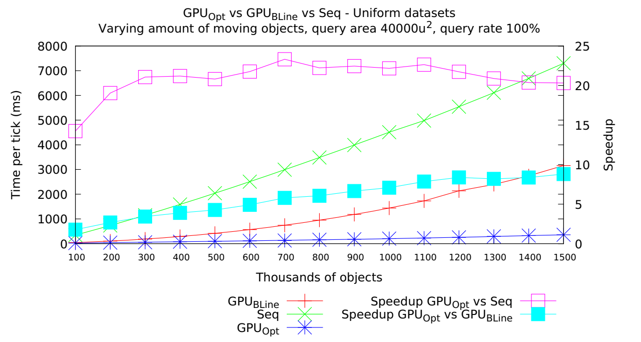

Fig. 5: Varying objects, running time per tick.

function of the number of objects, when the query size is fixed at $40k\ u^2$, and the query rate is fixed at 100%. While $\mathsf{GPU_{BLine}}$ has just a marginal advantage over Seq, $\mathsf{GPU_{Opt}}$ consistently outperforms both Seq and $\mathsf{GPU_{BLine}}$. The figure also shows how the speedup obtained by $\mathsf{GPU_{Opt}}$ stabilizes around 20x, even when the amount of moving entities becomes very big ($>$ 1M entities). This also shows the scalability of $\mathsf{GPU_{Opt}}$.

If a reader considers how the grid cells are dimensioned, s/he could argue that the relatively small query size, compared to the size of the spatial region, may favor $\mathsf{GPU_{Opt}}$. Thus, in the next batch of experiments we show the effects of varying the query area while keeping fixed the number of moving objects (at 700k) and the query rate (at 100%). In Figure 6

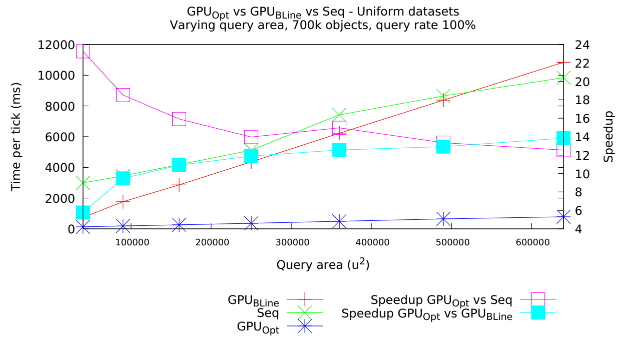

Fig. 6: Varying query area, running time per tick.

we can see how the advantage of $\mathsf{GPU_{Opt}}$ decrease when the query area increases. This hints that an optimal grid cell size indirectly allows us to find a balance between the average amount of joins per query actually computed, and a proper workload distribution over the different SMs of the GPU, by avoiding too many/few objects in some cells.

Due to space limitations, we do not show the plots of execution time vs. query rate. Obviously, the execution time depends linearly on the query rate. Moreover, if we increase the query rate, also the speedups obtained by both $\mathsf{GPU_{Opt}}$ and $\mathsf{GPU_{BLine}}$ with respect to Seq increases, since the number of (independent) computations per tick gets larger thus favouring GPU-based parallel solutions.

In order to assess one of the challenge this problem present, we now pass to study the size of the output, i.e., the result sets of the queries in each time tick. This output is larger than the input for typical settings, and it grows when $|P|$, $|Q|$, or the query area increase. Given a uniform spatial distribution of the objects in the the spatial area (sa), the output of any query approximatively contains a number of objects equal to:

$$\rho = |P| \cdot \frac{qa}{sa} \tag{3}$$

where qa is the query area. If we denote with ca the area of each cell, we can also compute the average amount of objects per cell as $\rho_c = |P| \cdot \frac{ca}{sa}$. Considered the spatial indexing used and the fact that all the queries are equally sized, we have that $ca = qa$ and thus $\rho = \rho_c$. The size of the cumulative output of all the queries Q in a tick thus contains a number of objects equal to:

$$|R| = \rho \cdot |Q| \tag{4}$$

Recall that for the experiments of Figure 5, the query area is fixed at $40k\ u^2$ while the query rate is fixed at 100% ($|Q| = |P|$). Thus, the size of the cumulative output (see Eq. 4) becomes quadratic in $|P|$, since $|R| = \rho \cdot |Q| = |P|^2 \cdot \frac{qa}{sa}$. For example, still referring to Figure 5, for $|P| = 100k$ we have that $|R| \approx 790k$, and for $|P| = 1.5M$ we have that $|R| \approx 17.850M$, whereas the value computed using the equations are, respectively, $800k$ and $18M$

Finally, we study how the various pipeline stages of $\mathsf{GPU_{Opt}}$ (see Section III-C) behave when we change either $|P|$ or qa. For simplicity we measure monolithically phases 1–3, considered their limited computational weights with respect to phases 4 and 5. When $|P|$ is varied, qa is fixed at $40k\ u^2$ and the query rate is 100% (thus $|P| = |Q|$), phases 1–3 exhibit a linear behaviour, while phases 4 and 5 exhibit a quadratic behaviour. This is expected, considered the Equations 1 and 2. More interesting are the experiments in which qa is varied in the $[40k\ u^2\ \ldots\ 640k\ u^2]$ range, $|P| = |Q|$ are kept fixed at 700k and the query rate is fixed at 100%. From Figure 7 we can observe that phases 1–3 have constant timings ($|P|$ and $|Q|$ are always the same), while phases 4 and 5 have a linear behaviour with respect to qa. This is explained by considering that the overall amount of containment tests is given by $\sum_{c \in \mathcal{C}_A} |Q^c| \cdot |P^c| = \frac{sa}{ca} \cdot 4\rho_c \cdot \rho = 4\frac{sa}{ca}\rho_c^2 = 4|P|^2\frac{ca}{sa}$ (since $\rho_c = \rho$, see Eq.3), where $\frac{sa}{ca}$ gives the amount of cells while the second term gives the average amount of containment tests per cell.

V. Related Work

The idea of using the abundant and cheap computational power offered by GPUs to boost spatial joins dates back to the era when GPUs did not offer real general purpose computing

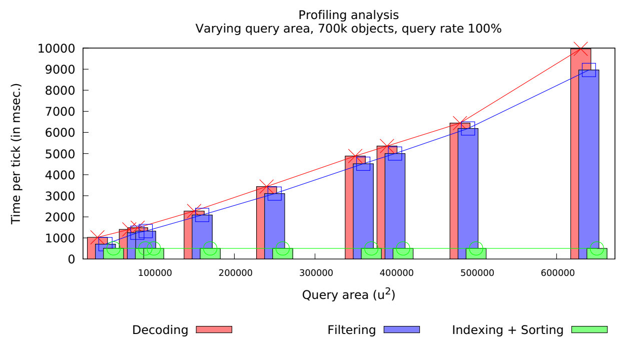

Fig. 7: Profiling analysis, varying query area, running time per tick.

capabilities [4]. As pointed out in an extensive review [2], the need for managing continuously incoming and evolving spatial data can be addressed by the usage of simple, lightweight (and, in many cases, throwaway) data structures, a fact that is particularly interesting when we consider the use of GPUs. However, it is crucial that data structures and algorithms contend effectively with skewed data distributions and avoid redundant spatial joins and bad workload distributions as much as possible. Recent studies [6], [7] show how uniform grid-based approaches like the one adopted in this work are particularly attractive when managing continuously incoming and evolving spatial data in main-memory multi-core settings. Other works consider the problems of building R-trees from scratch [13] and compute range queries using R-trees [3], [13] through an hybrid approach based on the combined use of CPU and GPU. We are unaware of any existing studies of repeated range queries on moving objects on GPUs. The most closely related work is focused on point-in-polygon joins [10]. Even if it would be possible to adapt parts of this approach to our scenario, there are key difficulties to consider: (i) the processed data is static, and the problem of managing continuous streams of updates and queries is not considered; (ii) applying this approach at each tick independently to deal with the dynamic scenario would be inefficient. A quadtree-derived data structure is used to partition the set of points as uniformly as possible, and the authors shows that this operation is quite expensive.

VI. Conclusions

We presented a novel and scalable techniques capable of computing on a GPU a massive number of repeated range queries over a huge amount of continuously moving objects. To achieve this goal we also introduced a new bitmap-based intermediate data structure that avoids memory access contention during parallel result output. The approach used and the data structure is general and can be reused in other contexts to improve the performance of operations that concurrently write interlaced list of results using coalesced memory access. We extensively tested the proposed algorithms to study their sensitivity to several parameters, and proved that there is a significant performance gain over a couple of baselines.

Skewed spatial distributions and variably sized queries may represent serious challenges for devising an efficient GPU-based solution to the problem of the repeated range queries on moving objects. The main issues are related to possible workload imbalance among GPU's SMs and cores. Many assumptions made for uniform distributions do not hold in such scenarios. In future works we aim to test different indexing methods and algorithms able to manage effectively such scenarios, while still fully exploiting the GPU computational power.

Acknowledgment

Research partially supported by the European Union under the ESF/COST COST Action IC0903: Knowledge Discovery from Moving Objects (MOVE, http://www.move-cost.info).

References

[1] Sowell B., Vaz Salles M., Cao T., Demers A., and Gehrke J. Indexing framework. http://www.cs.cornell.edu/~sowell/indexing/.

[2] Sowell B., Salles M.V., Cao T., Demers A., and Gehrke J. An experimental analysis of iterated spatial joins in main memory. Submitted.

[3] Yu B., Kim H., Choi W., and Kwon D. Parallel range query processing on R-tree with graphics processing unit. In *IEEE Conf. on Dependable, Autonomic and Secure Computing*, pages 1235–1242, 2011.

[4] Sun C., Agrawal D., and El Abbadi A. Hardware acceleration for spatial selections and joins. In *Proc. of ACM SIGMOD Conf.*, pages 455–466, 2003.

[5] Merrill D. and Grimshaw A.S. High performance and scalable radix sorting: a case study of implementing dynamic parallelism for GPU computing. *Parallel Processing Letters*, 21(2):245–272, 2011.

[6] Šidlauskas D., Ross K.A., Jensen C.S., and Šaltenis S. Thread-level parallel indexing of update intensive moving-object workloads. In *Proc. of SSTD Conf.*, pages 186–204, 2011.

[7] Šidlauskas D., Šaltenis S., and Jensen C.S. Parallel main-memory indexing for moving-object query and update workloads. In *Proc. of ACM SIGMOD Conf.*, pages 37–48, 2012.

[8] Dittrich J., Blunschi L., and Vaz Salles M.A. MOVIES: indexing moving objects by shooting index images. *Geoinformatica*, 15(4):727–767, 2011.

[9] Gray J. and Reuter A. *Transaction processing: concepts and techniques*. Morgan Kaufmann Publishers, 1993.

[10] Zhang J. and You S. Speeding up large-scale point-in-polygon test based spatial join on GPUs. In *Proc. of ACM SIGSPATIAL Intl. Wksp. on Analytics for Big Geospatial Data*, pages 23–32, 2012.

[11] Bentley J.L. and Friedman J.H. Data structures for range searching. *ACM Comput. Surv.*, 11(4):397–409, 1979.

[12] Hennessy J.L. and Patterson D.A. *Computer Architecture - A Quantitative Approach (5. ed.)*. Morgan Kaufmann, 2012.

[13] Luo L., Wong M.D.F., and Leong L. Parallel implementation of r-trees on the gpu. In *IEEE Conf. on Design Automation*, pages 353–358, 2012.

[14] Kornacker M., Mohan C., and Hellerstein J.M. Concurrency and recovery in generalized search trees. In *Proc. of ACM SIGMOD Conf.*, pages 62–72, 1997.

[15] NVIDIA. *CUDA C Programming Guide 5.0*. October 2012.

[16] Hong S. and Kim H. An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. *ACM SIGARCH Computer Architecture News*, 37(3):152–163, 2009.

[17] Sengupta S., Harris M., Zhang Y., and Owens J.D. Scan primitives for GPU computing. In *Proc. of ACM SIGGRAPH Symposium on Graphics Hardware*, pages 97–106, 2007.

[18] Brinkhoff T. A framework for generating network-based moving objects. *GeoInformatica*, 6(2):153–180, 2002.

[19] Lee V.W., Kim C., Chhugani J., Deisher M., Kim D., Nguyen A.D., Satish N., Smelyanskiy M., Chennupaty S., and Hammarlund P. Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. In *ACM SIGARCH Computer Architecture News*, volume 38, pages 451–460, 2010.