Massive Spatial Query on the Kepler Architecture

Yili Gong, Jia Tang, Wenhai Li*, Zihui Ye
State Key Laboratory of Software Engineering of China, School of Computer, Wuhan University
Wuhan 430072, Hubei, PR China
*Corresponding author: lwh@whu.edu.cn

***Abstract*—In this paper, we present an optimized framework that can efficiently perform massive spatial queries on the current GPUs. To benefit the widely adopted filter-and-verify paradigm from GPUs, the skewed workloads are first associated with certain cells in a scaled spatial grid, such that the following range verification cost against the massive spatial objects can be significantly reduced. Particularly on the Kepler architecture, we highlight a two-level scheduling method to exploit good data localities by developing a novel dynamic scheduling method. Based on this virtual warp-based scheduling method, groups of threads can compete for the unbalanced tasks to ensure good load balance. We conduct various of skewed workloads with different object positions and query distributions, to evaluate our optimized methods. Experimental results show that, as compared to the existing fixed-size allocation methods, the proposed adaptive scheduling strategies improve the query throughput by one order of magnitude.**

***Index Terms*—GPU; spatial join; virtual warping; preemption**

I. Introduction

In the last decade, the Graphics Processing Unit (GPU) has evolved from being a graphics-specialized processor into a programmable processor for massive parallelism. The immense computational power that GPU can offer accelerated the relevant researches, in both data-intensive fields [1] and computational-bound applications [2], to achieve tremendous performance in general-purpose computing. While there are a variety of optimisation techniques, building effective data-intensive engines for GPU processing depends on (a) *regularity* of memory accesses [3], and (b) *homogeneity* of the computing behaviors [4]. Generally combining hierarchically scheduling methods, a warp-based scheduler concentrates [5] on dispatching a bunch of threads onto the encapsulated Processing Units (PUs) [6].

As a typical scenario of data-intensive applications, the geo-information processing [7] has received a lot of attention in recent years. A classic geo-positioned context requires to answer variant queries over massive moving objects, generally subjected to some spatial constraints to guarantee the distance between the objects and the queries no larger than a certain degree. Based on a general-purpose index structure, the comparison cost of these queries can be reduced to a large extent [8]. It's of interests to shoe up variant position-rich applications, such as tourist services, mobile commerce, traffic control and logistics management, by constructing developed processing techniques on the emerging high-performance hardwares. To benefit the filter-and-verify paradigm, many optimisation techniques have been proposed specifically for multi-cores CPU platforms, regarding how to effectively improve memory efficiencies [9] and computing parallelism [10]. However, few researches concerned with how to apply such applications on the massive GPU cores, with thorough considerations to bridge the gap between these randomly skewed accesses and the *regular* and *homogeneous* requirements of GPUs.

In this paper, we present an optimized framework that can efficiently schedule massive spatial queries on the current G-PUs. In terms of a general distribution method, this framework associates a bunch of range queries with a fixed scale of spatial grid, such that the range verification cost against the massive spatial objects can be significantly reduced. Based on the Kepler architecture, we highlight a two-level scheduling method to exploit good data localities by conducting a novel preemption-based dynamic scheduling method. We notice that, this method regularizes the tasks across the virtual warps in terms of common atomic operations, it can thus be extended to the mainstream massive-cores platforms. Experimental results show that, as compared to the fixed-size allocation strategy, the proposed methods improve the query throughput by one order of magnitude.

II. Related Works

High-performance techniques have been widely introduced by the data-intensive paradigm over the years [9]. Besides the moving objects processing context studied in this research, we also notice the iterative computation for machine learning [2], merging operations in databases [1] and traversing over the hierarchical structures [4], [15]. With dramatic development of the GPU in recent years, we also witnessed growing interest in this kind of modern processors in many problem domains.

In the GPU architecture, massive cores are integrated in a chip to realize largest degrees of data-level parallelism, with wide vector (SIMD) units to drive these cores in SIMT execution model [2]. While many applications benefit from GPU, its excess requirements for synchronization accesses hinder the related engines from complex indexes [1] or consistency models [19]. The traditional operations, such as sort [14] and matrix computation [12], [17], can be substantially scaled up with a regular data layout.

In the cases where the computational model can be hardly regularized, we could resort to some effective block scheduling strategies [20] or their fine-gained improvements [19]. By providing artificial barrier synchronization, Lo et al [3] identified the program pattern to avoid the irregular accesses. The idea of DeNovo was used in [19] to exploit reuse of written data and

978-1-5090-4825-0/17/$31.00 ©2017 IEEE

synchronization variables across synchronization boundaries. Similar strategies were also employed in [20]. There are many other GPU-enabled techniques, such as dynamic parallelism [16] and atomic-free optimisations [13]. Our work in this paper is on the flatten query pattern, and we leave the applications of these techniques in the complex computing context as our future directions.

The general GPU scheduling techniques can hardly be directly applied to our research, where an index is necessary to prune the candidate spatial objects [7]. In this paradigm, we have witnessed some CPU-based parallel optimisations [8], [10]. On the current GPUs, Ward [11] directly dive the spatial join with their pruning algorithm, without any thorough consideration of synchronization requirements of the GPU. We will concentrate on how to efficiently drive general spatial query by deliberative scheduling optimisations.

III. Driving Spatial Join in GPU

This section gives an overview of the GPU execution model, with its advantages of the efficient TB scheduling over recent architectures. Its massive parallelism aspect motivates us to drive efficient queries based on the spatial partitioning method.

A. Hierarchical Viewpoint of GPU

As the demand for high-performance computing increases across many areas, industrial community continues to achieve extraordinarily powerful GPU computing architectures. For providing flexibly programming supports and massive parallelism, the GPUs generally offer groups of threads to drive the underlying processing units (PUs) to execute in parallel. By providing the `Barrier synchronization` across the PUs, a mechanism to stop executing at certain points can be achieved to ensure the PUs to reach these desired points before they go ahead with the following loads.

In the recent architectures, GPU programs generally consist of multiple kernels, each contains functions that can be executed by thousands of light-weight threads. These threads are launched as grid of thread blocks (TBs), so that one can assign all of its entire threads into a `Scheduler` to occupy groups of PUs. Based on a desired round-robin strategy, in each cycle, the `Scheduler` activates a certain number of threads in each TB. While there are various of techniques to drive massive PUs, very often each PU group and their encapsulated resources can be regularized by the warp-based scheduler. In each cycle, a certain number of warps are issued to execute concurrently. Through barrier synchronization and shared memory, a TB can thus be featured as a set of concurrently executing threads that can cooperate among themselves or become blocked due to the stalled cycles derived from the warp scheduling.

Let's take Kepler GK110 to illustrate how massive PUs are collaborated to perform a computational task in parallel. On a full GK110 implementation that is composed of 15 Streaming Multiprocessors (SMXs) and 64-bit memory controller, the entire PUs (Cores) are grouped into the physical SMXs, each consists of 192 Cores concurrently driven by four warps. We

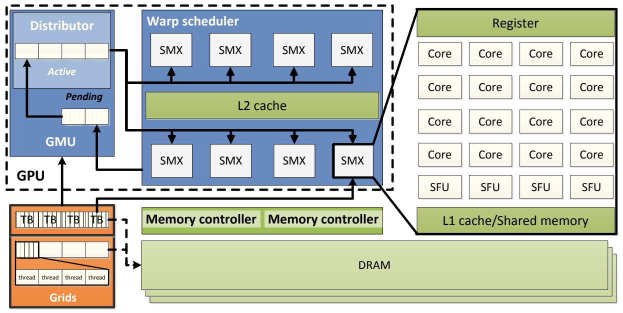


Fig. 1: Illustration of Kepler GK110 architecture, with grids of thread blocks (TBs) mapping onto their SMXs.

should notice that, in general data-parallel paradigms, each TB has a per-block shared memory space targeting at inter-thread communication, data sharing, and result sharing.

As given in Figure 1, a set of TB grids is delivered to a Grid Management Units (GMU), where each TB can be depicted as a desired functional kernel. In terms of a one-to-one mapping from the TBs to the SMXs, Kepler 110 allow kernels to launch task directly on GPU. By this way, the SMX scheduler dispatches the common kernel onto the entire SMXs in a round-robin fashion, with the TBs being activated in a TB ID increasing order. Based on the migration between the two queues, each active TB is dispatched onto a SMX to occupy the encapsulated resources in round-robin. These resources incorporate 64 KB of on-chip memory that can be configured as 48KB of shared memory with 16KB of L1 cache. Kepler 110 provides a quad warp scheduler for each SMX to select four warps each schedules 32 threads on two independent instructions in each cycle. It also expands the 64-bit atomic operations in global memory, such that the threads can issue read-modify-write operations on shared data structures concurrently.

B. Partition-based Spatial Data Organization

A spatial range query is to obtain the spatial objects that have the form of multi-dimensional points, polylines or polygons, so that these objects and the query condition satisfy some specific spatial relationships. Although there exists nine basic spatial relationships across the spatial data types, the spatial `overlap` is generally taken as the pivot to support the common spatial relationships. In the moving objects context, the objects mainly include massive spatial points, each is composed of a two-dimensional (2D) coordinates. By a range of maximal bounding box (MBR) of a query, the moving objects query is to generate the points that lie in (or meet with) a derived MBR. The advanced Nearest Neighbors (NN) query and its variants can be translated into a filter-and-verify workflow based on the `range query`.

Given a query range $r : (xmin, ymin, xmax, ymax)$ on a collection of objects $O = \{o : \langle id, pos(x, y)\rangle\}$, a näive strategy is to scan all the objects to check whether $o.pos$ lies in r, and return $o.id$ if it comes true or skip it otherwise. To reduce the complexity, an index is generally used to filter masses of objects that are outside the indexed entries with their MBR intersecting with r. In the concurrent execution on an in-memory environments, the spatial partitioning methods are superior to the point-clustering ones due to the small redundant entries. Although we study on the 2D grid structure, the related techniques can support higher dimensions as well.

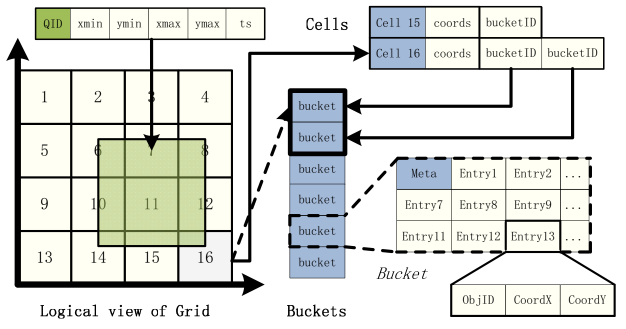


Fig. 2: A grid structure dividing the spatial domain into 4 × 4 cells, each maintains their objects by the fixed-size buckets.

As shown in Figure 2, our 2D grid orthogonally divides the spatial domain into the segments in the two spatial dimensions, resulting in a collection of disjoint `cells`, each has the form of *cell* : $\langle CellID, coords(xmin, ymin, xmax, ymax)\rangle$ with a list of bucketID attached. For clarity, we denote their $cellID$ as the numbers $1 - 16$ in the logical view of the grid. In each `cell`, the involved objects are maintained in a series of `buckets`, with their point positions inside an `Entry`. A `range query` r is translated into two stages, where the first stage is to calculate the `cells` that are intersecting with r, and the second stage compares the objects positions against r.

C. Query-Centered Spatial Join

There are plentiful researches regarding how to do efficient spatial queries on multi-core platforms. In GPUs, the straight-forward implementations usually results in poor spatial locality, workload balance and complexity control flow, making the query throughput suboptimal.

Due to the skewed workloads, it's difficultly to guarantee good localities for spatial queries, which results in suboptimal throughput in GPUs. In real-world scenarios, the skewness is derived from two folds. First, the moving points are substantially gathered at some regions, where the point densities are far more than the other regions. In addition, user interests often focus on some "hotspot" which is also the dense region of the spatial domain. Since the grid structure uniformly divides the spatial domain, the above-mentioned skewness may result in very unbalanced workloads across their logical executors.

In Figure 2, suppose each `bucket size` has a maximal capacity of 128, *cell* 15 has 100 moving objects, such that the $query$ (with its range intersected with *cell* $6-8, 10-12, 14-16$) needs to access one bucket in *cell* 15 as well as all of its 100 objects. In contrast, the cost on another cell *cell* 16, with 200 objects inside, doubles the cost derived from *cell* 15. If we drive the same $query$ on the two cells separately by two threads in a GPU, these two different capacities will most likely make the thread run on *cell* 15 stall half of its running time. Even worse, since we generally configure a TB on the consecutive cells, the corresponding threads need to access the dispersing linked buckets, which make these threads can hardly exploit the SIMT superiority of GPUs.

Algorithm 1: Dispatch queries based on a grid structure.

```
   Input: Query: query vector; qd: whether query driven
   Output: RCA: Requests associating cells and queries.
1  T_ID ← threadIdx + blockIdx * blockDim;
2  $i$ ← T_ID ;              // Index of the query vector
3  while $i < N$ do
4      $query$ ← Query[$i$] ;               // Get the query
5      cells ← CellsCoveredBy(query) ;           // Overlap
6      for cell in cells do
7          if qd then                        // Query driven
8              RCA[$queryId$].list.writeRequest($cell$);
9              RCA[$queryId$].meta.num++;
10         else                             // Cell centered
11             $rid$ ← atomicAdd(RCA[$cellId$].meta.num);
12             RCA[$cellId$].list[$rid$].writeReqeust($query$);
13         end
14     end
15     $i$ ← $i$ + blockNum * blockDim;          // Step ahead
16 end
```

To improve the thread-access localities, we propose a general dispatching strategy to support both the query-driven and cell-centered optimisation techniques in the following sections. Our motivation is to translate the näive round-robin execution of massive `range queries` into a two-stages join, where the first stage constructs a vector of requests mapping the objects to their intersected queries, so as to support the join in the second stage to batched answer these requests.

As shown in Algorithm 1, by using a TB length parameter blockDim (in line 1), the entire threads dispatch the queries based on a grid in parallel. We use a core list structure RCA to maintain the requests in the both strategies. If we use the query-driven processing method, RCA corresponds to the full query vector, with the associated cells in each element. In a coarse level, if a query overlaps a candidate cell (generated in line 5), the cellId will be written into the list field of the $queryId^{th}$ element in RCA. Since the entire threads handle a vector space of input queries with their stepping granularity blockIdx.x * blockDim.x (as shown in line 2 and 15), this distribution can exploit the SIMT superiority of GPUs. While in the cell-centered method, each *cell* in line 12 has a field $cell\ ID$, which denotes a index of global vector to maintain the cell list. In each element of this cell list, a header is maintained to redirect the objects requirement to a linked list of fixed-size buckets.

We generate the candidate cells by `CellsCoveredBy()`, which calculates the candidate cells of each range in terms of the spatial-partitioning grid structure. This direct computation doesn't depend on the hierarchical structures of the traditional indexes, so that the efficiency of the parallel threads can be guaranteed in GPUs.

IV. Cell-Driven Execution on Virtual Warp

This section highlights our cell-driven execution of a batch of spatial queries. It places two processing strategies on groups of threads to run the queries on massive partitioned objects.

A. Alternative Execution Strategies

Besides the widely-adopted query-driven strategy, a cell-centered processing method is presented in the virtual warp context. Our basic idea is to drive multiple queries related to a common cell in consecutive threads in a TB, so that the warp scheduler can dispatch these threads in the same cycle of each SMX. In addition, since the buckets in each cell is significantly larger than each single query, this cell-centered method ensures the sibling threads to handle the objects altogether, taking full advantage of the data locality based on *regular* object accesses.

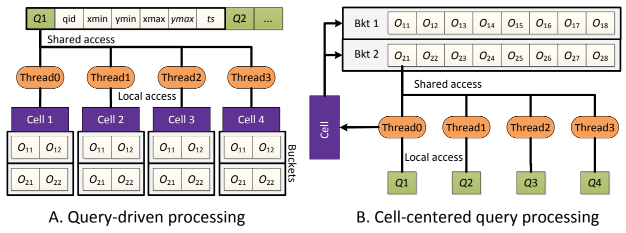


Fig. 3: Alternative strategies of spatial join on a cell-centered structure, with inverse access behaviors on queries and cells.

We demonstrate the both strategies in Figure 3, where a list of queries are executed by four threads with their candidate cells accessed in different behaviors. As shown in Figure 3A., once the four threads are active, the query-driven strategy shares a common query and extract the buckets in each cell, to verify whether the objects are within the range. In a high level, each thread needs to obtain their local cell to perform the verification against the query. Inversely, the cell-centered method (depicted in Figure 3B) shares a common cell and try to schedule the grouping threads to simultaneously handle the related queries that has been distributed by Algorithm 1. Once a thread make the candidate objects ready in a global space, all the in-group threads verify the positions of these objects against their private query. It's highly desirable that, as compared to the query-driven one, this cell-centered strategy can avoid the *irregular* access of chunks of object buckets. While in a scene where the queries is substantially skew, an extremely dense region could make its bounded "bottleneck" thread significantly slow, hindering the other sibling ones.

B. Grouping Threads by Virtual Warps

We next show how we improve the above-mentioned "bottleneck" threads by grouping them into `Virtual Warps`. While in the common spatial context, a long-term thread under a one-dimensional TB configuration can consume too many cycles because it needs to process the entire queries related to a cell. Alleviating the workloads of these "bottleneck" threads is crucial to decrease the stalled time of their sibling ones.

We implement the `Virtual Warps` by logically dividing the threads that belong to each TB into a set of subgroups, where each subgroup of threads share the instructions and hardware resources similar to what they can do among TB. With careful software considerations, such as the warp-level synchronization and homogeneous workload dispatching, we concentrate on providing a flexible micro-system to drive these grouping threads to cooperate with lightweight data access. By choosing one of these inner-warp threads as a controller, we can synchronize these threads as will. We abstract this `warp synchronization` as shown in Algorithm 2. For simplicity, we prepare a fixed warp size W_SZ as a global variable, such that each thread within a warp can be synchronized based on their warp identifier W_ID. As will show in the sequel, after all the inner-warp threads finish their current `subtask`, a `Virtual Signal` vector VS with a offset W_ID can be used to trigger these threads to execute the next round.

Algorithm 2: VW_synchronize(): Warp synchronization

```
1 atomicAdd(VS[W_ID]);
2 while VS[W_ID] % W_SZ != 0 do
3     nothing;
4 end
```

Another interesting question is how to produce homogeneous workloads, i.e., how to guarantee the balance of these inner-warp threads. In terms of `warp synchronization`, we call the workload between two consecutive synchronizations of each thread as a `subtask`. In a cell-centered `task` that is generated by Algorithm 1 in each element of RCA, we can regard all the overlapped queries against a same cell as homogeneous `subtasks`. Since a cell is shared by all of the inner-warp threads, the cost of comparing the entire objects to each related query has a constant complexity. If we were to run each query on a thread, the workload of these homogeneous `subtasks` can be easily guaranteed. Let's take Figure 3B as an example, the four threads have almost the same cost to join Cell (with its two buckets) with each query of $Q1 \sim Q4$.

C. Cell-Centered Join Implementation

So far, we present the basic ideas of the both join strategies. As the query-driven join can be easily realized by changing RCA into its $queryId$-derived form (as given in Algorithm 1), we focus on the cell-centered method in the remaining section and leave the comparison in Section VI.

We suppose a request list RCA is generated by Algorithm 1, with M cell-centered requests inside. We deliberate a three-level structure for the entire threads, respectively with their parameters of GPU grid size blockNum, block size blockDim and warp size W_SZ. In other words, we have blockDim / W_SZ warps in each TB, where each thread with its global identifier T_ID can determine its warp identifer by W_ID = T_ID / W_SZ as well as derive its local offset in a warp as W_OS = T_ID % W_SZ. Corresponding, the cell-centered join implementation can described as shown in Algorithm 3.

Algorithm 3: Cell-centered spatial join using a grid.

```
   Input: RCA: Request list with query area
   Output: QR: List associated queries and their results.
1  T_ID ← threadId + blockIdx * blockDim;
2  W_ID ← T_ID / W_SZ; W_OS ← T_ID % W_SZ ;
3  STRIDE ← blockNum * blockDim / W_SZ;
4  $i$ ← W_ID;                      // Corresponding to W_ID
5  while $i < M$ do
6      rca ← RCA[$i$];             // Warp-based acquisition
7      for $j$ from W_OS to rca.meta.num step W_SZ do
8          $query$ ← rca.list[$j$];      // Step within warp
9          for $object$ in cells[rca.meta.cellId].buckets do
10             if $object$.inRange($query$) then
11                 QR[$query$].writeObject($object$);
12             end
13         end
14     end
15     $i$ ← $i$ + STRIDE;                     // Step ahead
16 end
```

Once Algorithm 1 produces a RCA list, Algorithm 3 starts initializing the thread and warp parameters (respectively in lines 1 and 2). We divide the essential join into two levels, where the top level regards the warp-based scheduling, with a step size STRIDE to divide ahead the entire threads (of total blockNum of GPU grids, each has blockDim threads). In each warp, as shown in line 6, a request is obtained by the inner-warp threads in terms of their warp identifier. For each thread, a query related to a cell by the warp identifier (in lines 4 and 6) is located by the thread identifier in line 7. This cell-centered dispatching guarantees that, in terms of the common cell sharing across the inner-warp threads, the threads can join their local queries on the common buckets in a *regular* access behavior. The actual join work from line 9 to 13 compares the objects to the query range in line 10 based on the spatial coordinates in round-robin (line 7).

V. Preemption-based Warp Scheduling

This section presents how we provide better balance across the warps. Based on the cell-centered dispatching, it employs atomic operations to let the warps racing over the request queue RCA to avoid the stalled time of the short-term threads.

A. Motivation of Preemption

In the proposed strategy, each warp concentrates on executing the time-consuming spatial verification in terms of the query ranges and the pruned cells . As shown in Section IV-C, this cell-centered strategy ensures the load balance within each warp, but its fixed-size stepping (in line 15 of Algorithm 3) hinders the load balance across the warps. Even worse, one generally introduces a periodical synchronization for sake of *regular* memory accesses [16] or moving position freshment [7], which results in suboptimal throughput derived from the unbalanced workloads across different warps.

We introduce the atomic operation `atomicAdd`() to offer a racing strategy across the warps. Our motivation aims that, in the massive threads environments, dispatching new `tasks` dynamically to the warps that have obtained a scheduling cycle and competed its current `task` is helpful to reduce the stalled time. In addition, grouping the threads into warps can also reduce the racing intensity of this warp-level `preemption`. As compared to the parallel scheduling in Section IV with a fixed step size, this on-demand strategy relaxes the requirement of the even division of the entire warps.

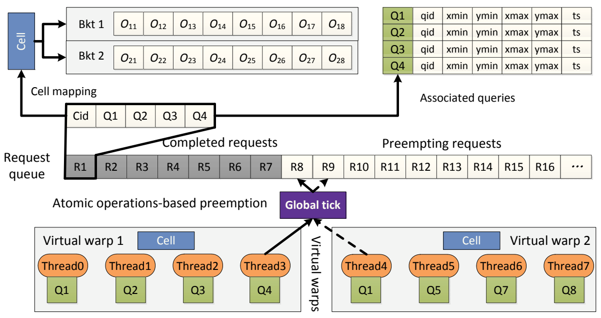

Cell
Bkt 1
O_{11} O_{12} O_{13} O_{14} O_{15} O_{16} O_{17} O_{18}
Bkt 2
O_{21} O_{22} O_{23} O_{24} O_{25} O_{26} O_{27} O_{28}
Q1 qid xmin ymin xmax ymax ts
Q2 qid xmin ymin xmax ymax ts
Q3 qid xmin ymin xmax ymax ts
Q4 qid xmin ymin xmax ymax ts
Cell mapping
Associated queries
Cid Q1 Q2 Q3 Q4
Completed requests
Preempting requests
Request queue
R1 R2 R3 R4 R5 R6 R7 R8 R9 R10 R11 R12 R13 R14 R15 R16 ···
Atomic operations-based preemption
Global tick
Virtual warp 1
Cell
Thread0 Thread1 Thread2 Thread3
Q1 Q2 Q3 Q4
Virtual warps
Cell
Virtual warp 2
Thread4 Thread5 Thread6 Thread7
Q1 Q5 Q7 Q8

Fig. 4: Virtual Warping preemption based on an atomic add.

As shown in Figure 4, in the proposed `preemption`-based scheduling strategy, the entire warps compete for the requests by a global `tick`. By conducting an atomic operation on this `tick`, the `head` of a warp tries to obtain a request when all of its sibling threads are free. In detail, the `tick` keeps an increasing index of the request queue, which monotonously differentiates the elements of the queue into two parts. In each time `tick` is increased by 1, the selected request will become a completed one since the entire warps will no longer choose its index in the future contention. The remaining structures, such as the cell mapping and the associated queries of each request, are the same as what we have presented in Section IV.

B. Dynamic Warp Scheduling

The dynamic warp scheduling can be realized by providing two inner-warp controllers, i.e., Virtual Signal vector (VS) and Notice Signal vector (NS). As presented in Section IV-B, each element of the VS is used to synchronize the inner-warp threads to collaborate on the cell-centered `tasks`. To share the request within a warp, a `head` thread of each warp is responsible for preempting on the global `tick` and writing its acquisition onto a shared memory region NS.

Algorithm 4 details our `preemption`-based warp scheduling method. Similar to Algorithm 3, it employs the cell-based join paradigm to highlight how this dynamic scheduling can be realized on the proposed join context. To share the above-mentioned vectors VS and NS within each warp, an initialization procedure is triggered by the `root` thread of the total blockNum * blockDim threads. It initializes `tick` S and allocates the two shared vectors VS and NS with their vector length as the warp number $\lceil$blockNum * blockDim / W_SZ$\rceil$. We omit its formal details due to page limitations.

Very different from Algorithm 3, this `preemption`-based method employs an adaptive warp scheduling instead of the previous fixed-size stepping. As shown in line 6, the upcoming counter of `tick` S is competed by all the `heads`. In each

Algorithm 4: Warp preemption by the atomic operations

```
Input: RCA: Request list with query area.
Output: QR: List associated queries and their results.
T_ID ← threadIdx + blockIdx * blockDim;
W_ID ← T_ID / W_SZ; W_OS ← T_ID % W_SZ;
initialize(S, VS, NS);                    // Initialization
while true do
    if W_OS == 0 then                     // The head of a warp
        $s$ ← atomicAdd(&S, 1);            // Preempt on S
        if $s < N$ then
            NS[W_ID] ← $s$;                // Share $s$ in warp
        end
    end
    VW_synchronize();                     // Warp synchronization
    if $s \geq N$ then
        break;
    end
    rca ← RCA[NS[W_ID]];                  // Get request
    join on rca as lines 7∼14 in Algorithm 3;
end
```

round, after a head (denoted by line 5) acquired a counter, its sibling threads (synchronized by Algorithm 2) begin to access the acquired request after a warp synchronization in line 11. In each warp, these sibling threads share a common request directed by the index NS[W_ID], so that they can handle their corresponding queries in round-robin, as given in lines 7∼14 of Algorithm 3. This method focuses on the global contention towards tick S across all the warps. It will terminate at line 12 when all the requests are processed.

C. Analysis on Join Workloads

It's clear here that, in our proposed scheduling framework, massive spatial range queries are conducted on a two-stage executing context. We employ a round-robin TB scheduling in the first dispatching stage, to associate the range queries with their related cells. While in the second verification stage, a two-level scheduling method is proposed to guarantee load balances as well as *regular* memory accesses. By analyzing the cost of the staged workloads, we next show why we arrange the entire threads in such different ways in the both stages.

In the dispatching stage, the computing model can be described as the homogeneous workloads to associate the queries with their overlapping cells. As discussed in Section III-C, the computation cost of the overlapping cells is trivial since it runs on-the-fly, without extra memory access besides the algorithmic input and output. We can respectively formalize the *D*ispatching **C**omputation and **M**emory access costs as

$$\mathbf{C}(D) = \sum\nolimits_{i=0}^{N} |\texttt{cells}(Query[i])| \text{ and} \tag{1}$$

$$\mathbf{M}(D) = |Query| + \mathbf{C}(D) + \hat{\mathbf{C}}(D), \tag{2}$$

where $|\texttt{cells}|$ denotes the cell number and $\hat{\mathbf{C}}(D)$ denotes the atomic operating cost of total $\mathbf{C}(D)$ memory accesses. Since Algorithm 1 doesn't use thread synchronization for each batch of queries, it can avoid the stalled time at the cost of *irregular* memory accesses. In addition, the cell-based dispatching will bring the memory contention cost due to the introduction of the atomic operations. Our experiments show that, as compared to the significantly heavy workloads of the verification stage, this contention cost can hardly impact the total throughput.

In the actual join stage of the cell-centered processing, each warp controls their working threads to collaboratively perform the queries against a common cell. This results in the access cost quite different from the total overlapping cardinality as given in Equation 1. Let's suppose each cell has at least one query to be related, the total **C**omputing and **M**emory access costs in the *V*erification stage using the cell-centered strategy can be approximatively expressed as

$$\mathbf{C}(V) = \sum\nolimits_{i=0}^{N} \sum\nolimits_{cell \in \texttt{cells}(Query[i])} \|cell\| \text{ and} \tag{3}$$

$$\mathbf{M}(V) = \sum\nolimits_{cell \in \texttt{CELLS}} |\texttt{buckets}(cell)| + \left|\widehat{\texttt{CELLS}}\right|, \tag{4}$$

where $\|cell\|$ denotes the total object number within a cell.

As analyzed in the previous sections, in the proposed cell-centered processing framework, the total computing cost given in Equation 3 can be evenly shared by the entire threads based on the deliberative warp scheduling methods. It's naturally much more superior to its query-driven counterpart when the objects and queries are both skewed. The extra atomic operations in the cell-centered strategy, with their atomic operating cost $\left|\widehat{\texttt{CELLS}}\right|$ given in equation 4, can bring significant access cost when massive cells are conducted. We will analyze these interesting singular points in our experiments.

VI. Experimental Evaluation

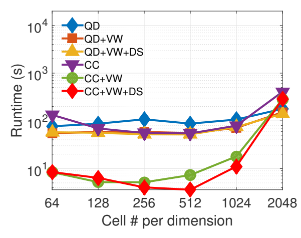
To evaluate the proposed methods, we conducted our experiments on a data set by using (and extending the object cardinality of) the MOBenchmark [7], with up to 100 millions of moving objects in a square region of 100km × 100km. To measure the availability of the methods in skewed workloads, we selected a fixed number of points as the "hotspots", with their query centers or moving objects normally distributed within a certain radius. We denoted the data set with desired numbers of "hotspots" as the Gaussian data, with the queries concentrating on some "hotspot regions" the Gaussian query. This synthetic roadmap, with controllable balancing factors on the moving objects and queries, has been widely used to measure the availability of the related optimisations [7], [11].

We applied two levels of scheduling methods Virtual Warps (VW) and Dynamical Scheduling (DS, based on preemption) orthogonally onto the two join strategies, i.e., Query-Driven (QD) and Cell-Centered (CC). We named the related methods as the join strategies plus their (compositional) scheduling methods. For example, CC+VW+DS stands for Cell-Centered join optimized by Virtual Warp and Dynamical Scheduling.

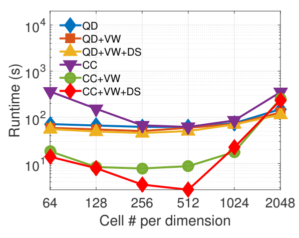
We conducted our experiments on a host Server with dual Intel Xeon E5-2650(2.4GHz) CPUs and 16GB memory. We executed all the GPU programs on a NVIDIA (Kepler) Tesla K40c GPU. The programs were compiled by CUDA-8.0 and run on a 64-bits CentOS-7.2 with linux kernel-3.10.0. We configured 10 blocks each consisted of 1024 threads for the baseline methods, which was the best parameter for our QD and CC implementations. The virtual warp-based methods scheduled each virtual warp together, which may regard the thread groups as its scheduling units. We thus increased the block parameters up to 40, each was composed of 32 warps $\times$ 32 threads, to evaluate the remaining four optimized methods.

A. *Cell Regularization*

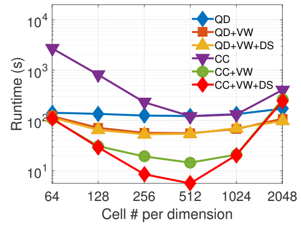
We conducted the first experiment with 1 millions queries, with their 10 millions objects dispersing over desired numbers of "hotspots". It's well-known that the cell granularity heavily impacts the query efficiency. We used 0.064 as the range to cover random squares, with their side lengthes of $0.064 \times 100km = 6.4km$, or equally a square of $40.96km^2$. By varying the cell numbers, we measured the running time (in second) as plotted in Figure 5, where the four sub-figures gave the same experiment on different scale of data skewness, respectively in `uniform` distribution and `Gaussian data` within 10, 100, 1000 gaussian cores. The both coordinate axes are logarithmic, which made some differences too small to see.



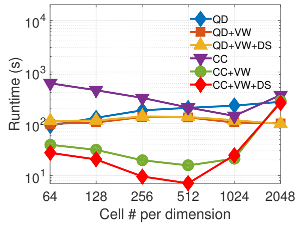
(a) Query on uniform data



(b) Query on 10 gaussian cores



(c) Query on 100 gaussian cores



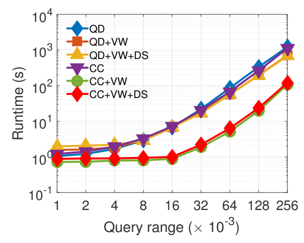
(d) Query on 1k gaussian cores

Fig. 5: Query efficiencies in different granularities of grid cell.

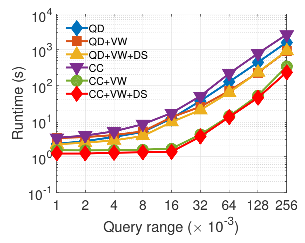
All the four results revealed that, with the increasing cell granularities, the QD family exhibited stable tendencies of running time. The VW and DS-intensified versions outperformed QD if we chose a larger cell number in very skewed dataset, i.e., the QD+VW and QD+VW+DS respectively saved 81% and 83% running time as that of QD in Figure 5c. While in the CC family, a significant saddle tendency was demonstrated in all the four data sets. As analyzed in Section V-C, too elaborate cells will result in huge numbers of cells, which will definitely increase the scheduling rounds in our CC strategy. As shown in the figures, the CCs exhibited rare superiorities as compared to the QDs in a cell granularity of 2048. Reversely, when we chose extremely small granularities, i.e., 64, the weak pruning power from the spatial grid structure depressed the superiority of CCs, which made CCs very close to QDs. If the both optimized versions were run on their best granularities, *CCs outperformed QDs by one order of magnitude, i.e., in granularity 512 of CCs. Given the best granularity, CC+VW+DS showed 7X superiorities to CC+VW.*

B. *Query Range*

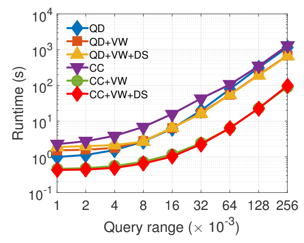
This experiment fixed the cell granularity with 128×128, to measure different skewness of both 1 million queries and 10 million objects. By varying the query range exponentially from 0.001 to 0.256, we plotted the results as shown in Figure 6.



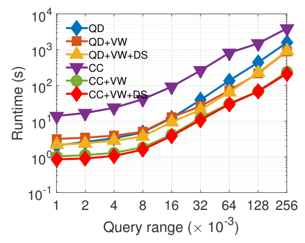
(a) Uniform query on uniform data



(b) Uniform query on gaussian data



(c) Gaussian query on uniform data



(d) Gaussian query on gaussian data

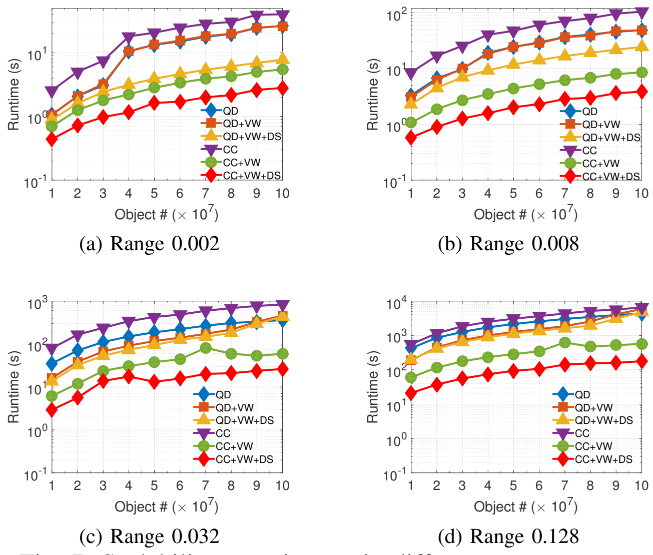
Fig. 6: Query efficiencies by varying the query ranges.

We could get better results of CCs by choosing the best cell granularity, such as 512 in Figure 5. Even in a moderate granularity 128, as revealed in Figure 6, the both optimized versions on CC were significantly better than their QD-based competitors. In Figure 6a and 6c, the formers were at most 11X better than their competitors. Due to the balanced `subtasks` in CCs in the moderate granularity 128, in this experiments, CC+VW+DS was not such superior to CC+VW. However, the both versions saved more than 90% running time of that of their original CC. We also noticed VW+DS optimisations, in both QD and CC cases, were increasingly superior to their baselines with the increase of the query skewness and range.

C. *Scalability*

Our last experiment used 100 `Gaussian cores` of moving objects, with the cardinalities varying from 10 millions to 100 millions. We respectively employed the skew query ange 0.002, 0.008, 0.032, 0.128 to measure the scalability of the proposed methods. With a fixed granularity 256, Figures 7 plotted the running time on the four query workloads.

It clearly revealed that, using preemptions on the both baselines can introduce significant benefits due to its balancing



(a) Range 0.002 (b) Range 0.008

(c) Range 0.032 (d) Range 0.128

Fig. 7: Scalability experiments in different query ranges.

considerations across the different scheduling levels. Very different from what we have observed in Figure 6, a larger cell granularity 256 in this experiment made CC+VW+DS nearly save 70% running time of that of CC+VW. *Interestingly, QD+VW+DS was on average 10X faster than QD+VW when the object cardinalities were beyond 20 millions.* One reason of this result could be ascribed to the density insensitivity of the QD strategy. In QDs, each warp concentrated on processing a `range query`, with their related cells being separately handled by the sibling threads; and this held in CC vice vasa. Since the cells related to the queries were highly different, the extremely dense regions could hinder the short-term warps in the both VWs. These unbalanced queries in DSs, however, were performed by the entire warps in round-robin, without introducing further "idle" warps. Even impacted by the small preempting cost, the both DSs was 5X more superior to VWs. In addition, due to the inner-warp balancing considerations of CCs, the both optimized CCs (CC+VW and CC+VW+DS) were 10X better than their competitors on QDs.

VII. Conclusions

This paper presents an optimized framework that can efficiently schedule massive spatial queries on current GPUs. The entire queries are first associated with a spatial grid to reduce the actual join cost. We highlight a two-level scheduling method to exploit good data locality by a novel scheduling method, so that groups of threads can effectively compete the unbalanced tasks. Experimental results, covering 100 millions of skewed moving objects and queries, show that the proposed strategies improve the spatial query throughput by one order of magnitude as compared to the existing methods.

VIII. Acknowledgments

This work was sponsored by the National Science Foundation of China, grant 61572373, 61100020, 60903035, the National High Technology Research and Development Program (863 Program) of China (No. 2013AA12A301), and the independent project funding of the State Key Lab of Software Engineering, Wuhan University (No. SKLSE-2015-A-04, SKLSE-2015-A-05). Besides the reviewers, the authors would like to thank to High Performance Computing Center of Wuhan University for the platform provision.

References

[1] T. Karnagel, R. Mueller, and T.M. Lohman. Optimizing GPU-accelerated Group-By and Aggregation. In *International Conference of Very Large Data Bases (VLDB)*, pp. 13-24, 2015.
[2] A. Yazdanbakhsh, J. Park, and et al. Neural acceleration for gpu throughput processors. In *Proceedings of the 48th International Symposium on Microarchitecture (MICRO-48)*, pp. 482-493, ACM, 2015.
[3] S.H. Lo, C.R. Lee, and et al. Improving GPU Memory Performancewith Artificial Barrier Synchronization. In *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 9, pp. 2342-2352. IEEE, 2014
[4] F. Busato, and N. Bombieri. BFS-4K: an efficient implementation of BFS for kepler GPU architectures. In *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 7, pp. 1826-1838. IEEE, 2015.
[5] S. Hong, S.K. Kim, T. Oguntebi, and K. Olukotun. Accelerating CUDA graph algorithms at maximum warp. In *ACM SIGPLAN Notices*, vol. 46, no. 8, pp. 267-276. ACM, 2011.
[6] J. Liu, J. Yang, and R. Melhem. SAWS: Synchronization aware GPGPU warp scheduling for multiple independent warp schedulers. In *Proceedings of the 48th International Symposium on Microarchitecture (MICRO-48)*, pp. 383-394, ACM, 2015.
[7] D. Sidlauskas, D. Saltenis, C.S. Jensen. Parallel main-memory indexing for moving-object query and update workloads. In *Proceedings of the 2012 ACM International Conference on Management of Data (SIGMOD)*, pp. 37-48, ACM, 2012.
[8] J. Dittrich, L. Blunschi, M.A.V. Salles. Indexing moving objects using short-lived throwaway indexes. In *Advances in Spatial and Temporal Databases*, pp. 189-207, 2009.
[9] V. Leis, A. Kemper, T. Neumann. Exploiting hardware transactional memory in main-memory databases. In *2014 IEEE 30th International Conference on Data Engineering (ICDE)*, pp. 580-591, IEEE, 2014.
[10] D. Sidlauskas, D. Saltenis, and et al. Trees or grids?: indexing moving objects in main memory. In *the 17th ACM SIGSPATIAL*, pp. 236-245, ACM, 2009.
[11] P.G. Ward, Z. He, and et al. Real-time continuous intersection joins over large sets of moving objects using graphic processing units. In *The VLDB Journal*, vo. 23, no. 6, pp. 965-985. 2014.
[12] Y. Nagasaka, A. Nukada, and S. Matsuoka. Cache-aware sparse matrix formats for Kepler GPU. In *20th ICPADS*, pp. 281-288, IEEE, 2014.
[13] R. Nasre, M. Burtscher, and et al. Atomic-free irregular computations on GPUs. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units (GPGPU)*, pp. 96-107 ACM, 2013.
[14] I. Tanasic, L. Vilanova, and et al. Comparison based sorting for systems with multiple GPUs. In *the 6th Workshop on General Purpose Processor Using Graphics Processing Units (GPGPU)*, pp. 1-11, ACM, 2013.
[15] C.K. Kim, C. Jatin, and et al. FAST: fast architecture sensitive tree search on modern CPUs and GPUs. In *SIGM)*, pp. 339-350, ACM, 2010.
[16] J. Wang, R. Norm, A. Sidelnik, and et al. Laperm: Locality aware scheduler for dynamic parallelism on GPUs. In *the 43rd ISCA*, pp. 585-595, IEEE, 2016.
[17] R. Zheng, W. Wang, and et al. GPU-based multifrontal optimizing method in sparse Cholesky factorization. In *2015 IEEE 26th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pp. 90-97, IEEE, 2015.
[18] S. Collange, M. Joldes, and et al. Parallel floating-point expansions for extended-precision GPU computations. In *2016 IEEE 27th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pp. 139-146, IEEE, 2016.
[19] M.D. Sinclair, J. Alsop, and S.V. Adve. Efficient GPU synchronization without scopes: Saying no to complex consistency models. In *Proceedings of the 48th International Symposium on Microarchitecture (MICRO-48)*, pp. 647-659, ACM, 2015.
[20] S.C. Xiao, and W.C. Feng. Inter-block GPU communication via fast barrier synchronization. In *International Symposium on Parallel & Distributed Processing (IPDPS)*, pp. 1-12, IEEE, 2010.