ABSTRACT
Contact tracing is gaining its importance in controlling the spread of COVID-19. However, the enormous volume of the frequently sampled tracing data brings major challenges for real-time processing. In this paper, we propose a GPU-based real-time contact tracing system based on spatial proximity queries with temporal constraints using location data. We provide dynamic indexing of moving objects using an adaptive partitioning schema on GPU with extremely low overhead. Our system optimizes the retrieval of contacted pairs to match both the requirements of contact tracing scenarios and GPU centered parallelism. We propose an efficient contacts evaluation mechanism to keep only the spatially and temporally valid contacts. Our experiments demonstrate that the system can achieve sub-second level response for large-scale contact tracing of tens of millions of people, with two magnitudes of performance boost over CPU based approach.

CCS CONCEPTS
• Information systems → Location based services.

KEYWORDS
contact tracing, moving objects, GPU

GPU-based Real-time Contact Tracing at Scale

Dejun Teng
Stony Brook University

Akshay Nehe
Stony Brook University

Prajeeth Emanuel
Stony Brook University

Furqan Baig
Stony Brook University

Jun Kong
Georgia State University

Fusheng Wang
Stony Brook University

1 INTRODUCTION
As of June 2021, COVID-19 has caused more than 170 million infected cases and 3.5 million deaths globally
1. With the progress of massive vaccine administration, the population is gradually being immunized. However, with inequality of vaccine access across countries [14], and reluctance of vaccination [12], it is highly likely we will live with COVID-19 for a long while, where infections will happen frequently. Contact tracing [11] is an essential approach to identify people who have had contact with COVID-19 positive people for potential quarantine or early interventions. That is, when an infected individual is identified, all the people physically contacted with the individual after a past time point need to be identified. In the past, contact tracing was mainly conducted by manual epidemiological investigation, and the traces of individuals were collected according to the descriptions. With the wide availability of smartphones, it is now possible to identify contacts automatically through proximity detection - how close a person was with another person in space, with temporal constraint - is there sufficient time for the two persons to meet. Common techniques include bluetooth-based methods and location-based methods [23].

Bluetooth-based methods rely on sharing signals between devices with Bluetooth installed to determine contacting between individuals [16]. The contact information is then shared with the stakeholder (e.g., the government) on demand only for the purpose of tracing potentially infected individuals. Bluetooth-based methods need dedicated software like PEPP-PT [40], TraceTogether [13], FluPhone [28], Alhosn [1] among others, or hardware [1, 13] to detect contacts. Apple and Google also released their Bluetooth-based solution at the OS level [2]. However, as the Bluetooth-based methods rely on the Received Signal Strength Indication, or RSSI, to estimate the distances between objects, it has been reported that a large portion of false positive cases will be retrieved [5, 25].

Location-based methods, on the other hand, retrieve contact information from the locations of the people. One location-based method in practice is to identify if people appear in specific locations by manual reporting. Another method is to use a scanner representing a location to scan a QR code named “health code” [8] from a user’s smart phone when entering the location. When an infected individual is identified, all the people who showed up at the same location will be identified for further investigation. Both methods have locations coarsely defined, thus large amounts of false positives will be generated. With the rapid availability of GPS-enabled smart devices and the continuing improvement of satellite based localization systems and in-door location systems, the locations of people can be detected in high geospatial accuracy, up to tens of centimeters [47], in particular for static sessions, centimeter level accuracy can be achieved with smart phones [41]. In this paper, we assume all locations can be well detected in good accuracy. Based on that, we envision that if the locations of all people can be sampled periodically in real-time, all the people within a certain distance from an infected individual at any time point can be retrieved. The query can then be propagated until all potential infected individuals are retrieved.

There are two approaches to manage traces of people for contact tracing: a decentralized approach logs the traces of people on their own devices (e.g. SafePaths [22]), and a centralized approach
reports the locations of people to a central node (e.g., South Korea’s contact tracing [29]). While the privacy may be better preserved with the decentralized approach, it has been reported that the effectiveness of the decentralized approach is much lower than the centralized approach [16]. The centralized approach works well for small population with large sampling interval. However, for a pandemic, the number of people to be tracked could be in millions or tens of millions for a relatively enclosed region such as a city, and the locations of people need to be sampled frequently, e.g., sub-second level. As a result, the volume of the raw trace data collected can be enormous. For instance, for a city with a population of 10 million and the location of each person being sampled every second, as discussed later, more than 10TB data will be collected daily, and more than 170TB will be collected every 14 days. Meanwhile, since most times a person will likely stay alone or only have contact with a specific set of people for enough time, it is possible to significantly reduce the data volume through real-time filtering.

A contact tracing data management system is essential to manage and detect contacts between individuals and infected individuals with spatial and temporal constraints from a large population, ideally in real-time, for earliest followup and interventions. This will require adaptive indexing of a large number of moving people, high throughput spatial-temporal evaluation of contacts among a large population, and effective filtering of contact pairs, all at extreme scale. In this paper, we propose GLINT, a GPU-based Real-time ConNect Tracing system that can achieve sub-second response for contact tracing data processing for a population at tens of millions scale. GLINT provides a fully GPU compatible pipeline, including GPU compatible elastic dynamic data partitioning, GPU-optimized contacted objects refining, and GPU based Spatial-temporal evaluation of contacts, which fully takes advantage of the massive parallelism of GPGPU with high throughput and low cost. Our contributions are summarized as follows:

- We propose a dynamic partitioning schema with updates at leaf-node level to index the moving objects on GPU with extremely low overhead;
- We boost the efficiency of retrieving and pruning contacted pairs with novel techniques including filtering-aware partitioning and refinement unrolling, ideal for the GPU computation architecture;
- We utilize a GPU-based hash table to achieve effective spatial-temporal evaluation of contacted pairs;
- We evaluate GLINT and provide guidelines on tuning the system to achieve optimized performance.

The rest of the paper is organized as follows. The background knowledge of contact tracing and GPU programming is given in Section 2, followed by the related works in Section 3. The designing details of GLINT are shown in Section 4. Section 5 demonstrates the efficiency of GLINT with experiments and discusses how it can be tuned to achieve maximum performance for given scenarios. The paper is concluded with Section 6.

2 BACKGROUND

Spatial Data Placement for Contact Tracing: Contact tracing is a complex query that identifies all the pairs of people falling under certain criteria from a dataset with the traces of all the people who are constantly moving. One common criterion is that one person contacts another person at a certain time point with the distance, Euclidean distance in general, under a threshold. Therefore, the contacted person of a target person at one time point can be retrieved with a typical within query with a traditional spatial data management system (SDBMS). Shirani-Mehr et al. studies the indexing and placement methods of large scale contact data [36], to facilitate spatial and temporal locality of the contact dataset by storing the coordinates of objects for different time points on disk into grid cells. Each of those grid cell holds the data records lying in one specific square for a specific period of time. When a contact tracing query is conducted starting from one specific object at one time point, only the data records for referred grid cells need to be loaded from disk and processed in memory, thus the overall I/O cost is minimized.

Temporal Constraints of Contact Tracing: On the other hand, in the case of tracing the spread of an infectious disease, one person may get infected only after exposure to an infected person for a long enough period of time. Fig. 1 lists two typical contact scenarios. In the "instant exchange" case, a transferring is complete immediately after two people contact, while in the "transfer delay" case, a transferring is complete only when two people keep contacting for a minimum continuous period of time [38]. In this paper, we assume that a contact is valid only when the duration of the contact is longer than a minimum contact time. Strzheletska et al. proposed to preprocess the raw traces [38] before conducting contact tracing queries. During the preprocessing, all contacts are retrieved and the ones with duration less than a minimum threshold will be pruned. While this can significantly reduce the overall data volume and increase the query performance, it also brings significant workload to preprocess the traces. In the scenario of disease control, the sampled locations of people for one time point need to be processed in real time before next round of sampling is conducted. However, the computation load for such a large amount of data could be difficult to handle in real time. This motivates us to address this challenge with General Purpose Graphic Processing Units (GPGPUs) [27].

GPGPU Computing Architecture: GPU achieves single instruction multiple data (SIMD) by executing massive amount of threads in parallel. Each GPU is equipped with multiple streaming multiprocessors (SM), and each streaming multiprocessor manages a fixed number of cores (e.g., 64 cores on NVIDIA RTX 20280 Ti). From the programming perspective, all the threads are organized in a hierarchical structure of grids, blocks, and threads. As for the latest GPU architecture, each block contains up to 1,024 threads, and every 32 threads in the same block are packed as a warp. All the threads in the same block are executed by the cores of the same
streaming multiprocessor, and all the threads in the same warp run the same instruction while being executed. For a conditional statement with different outcomes for the threads in the same warp, the execution paths for both outcomes need to be executed by all the threads. As a result, conditional statements will significantly degrade the performance of a GPU program.

**GPU Programming Data Structures:** There exist multiple GPU programming frameworks like CUDA [27] and OpenCL [18]. While we focus on the CUDA programming framework in this paper, the fundamental methods can also be applied to other frameworks. CUDA supports atomic operations on 32 or 64 bit words. Locks can be implemented by letting threads compete for writing to the same memory address, but the performance of using locks to protect critical operations on complex data structures is low and it counteracts the efficiency of GPU in processing data in parallel.

On the other hand, lock-free atomic operations over complex data structures like list, queue, stack, and hash table can be achieved if they can only be pushed or popped by all the threads during one execution round. Taking stack as an example, an integer counter is maintained and updated by all the threads, and when one thread need to push an entry to the stack, the counter will be atomically incremented by one with the `atomicInc` function and the original value of the counter is returned. The returned counter value will be used to calculate the target address in the stack and the data can be copied into the stack safely. Similarly, when popping out an entry from the stack, the counter will be decreased by one with the `atomicDec` function. As the stack is pushed or popped only during one round of execution, no competence will be invoked between different threads. Meanwhile, fully-functional stack can be implemented in block-level as CUDA supports synchronization between threads in the same block. A hash table can be implemented by letting threads compete for buckets with the `atomicCAS` (compare and swap) function. In this paper, all those data structures will be facilitated to achieve real-time contact tracing data processing on GPU.

### 3 RELATED WORK

It is crucial to make sure user privacy is well preserved in all contact tracing systems. 7 principles of designing a privacy-preserved contact tracing system is given in [33]. A block-chain enabled contact tracing solution is proposed in [45]. REACT, short for REAL-time Contact Tracing, is a comprehensive contact tracing solution [9]. Specifically, REACT enhances the privacy of existing mobile tracking by giving the user options to select frequency and precision of location sampling. A practice of traces sharing between individuals and government officials in South Korea is introduced in [29]. In general, it is a good practice to collect traces data anonymously with encrypted identity information which can only be decrypted by the user themselves or a trusted third party.

Indexing moving objects is widely studied. One typical query on moving objects is continuous range query which retrieves the objects that are covered by a fixed or moving query window in a time frame. Time-parameterized R-tree (TPR-tree), a R*-tree [15] variant, is proposed to indexing objects considering their movements [34]. In TPR-tree, an object is represented by both of its position in the past and its velocity. TPR = tree proposes improved construction algorithms over TPR−tree and achieves higher query performance[39]. B^4−tree further improve the efficiency of the time-parameterized range query and kNN query by modifying B^+−tree with time-parameterized feature [17]. Self-Tunable Spatio-Temporal B+-Tree (ST^2B−tree) partitions the space referring to certain points [7]. A voronoi graph is then generated with those reference points and each voronoi cell is partitioned with independent grids. Scalable Incremental hash-based Algorithm (SINA, for short) takes an hashing method to map the moving objects to moving queries which achieves efficient continuous query when both the objects and query windows are moving [24]. [10] takes an approach to generate short-lived index over snapshots of the object locations trading off with staleness of results. [26] takes advantage of the velocity skew of objects to reduce the movement of objects from 2 dimension to 1 dimension which reduces the complexity of indexing and querying with a tradeoff of correctness. Based on continuous range query, continuous intersection join can be implemented by defining each query window as the MBRs of all the objects in one dataset [48].

TPR-tree and its variants improve the indexing and querying performance for cases that the objects are constantly moving but are not suitable to be implemented on GPU as they require large amount of locking operations and dynamic memory allocations [43]. [37] and [43] implement the continuous range query and continuous intersection query over moving objects using GPU and facilitate grid cells to index the moving objects. However, indexing with grid cells is only feasible when the space is small and the objects are evenly distributed. On the other hand, facilitating GPU to improve the performance of indexing and querying static spatial data with tree-like spatial indexes like R-tree is also widely studied. Techniques like shared stack [21], hybrid DFS/BFS [46], and dynamic launching [31] are introduced to improve the efficiency of tree-like spatial index retrieval. G-tree is introduced in [19] for processing spatial data in hyper dimensions using GPU. [32] proposes a GPU-based solution to achieve efficient graph query in parallel which is not the focus of this paper. To the best of our knowledge, there is no existing GPU-based system on contact tracing data processing.

### 4 SYSTEM DESIGN

We will first present an overview of the GLINT system.

#### 4.1 Overview

The goal of GLINT is to retrieve the valid contacts from the locations of a set of objects which are continuously updated within a certain interval. At a specific time point, for each object, the objects it contacts can be retrieved with a within query by retrieving all the objects whose distance from it is smaller than a threshold, or the contact distance (contact_dist). Within queries are commonly conducted with a filter-and-refine paradigm [6]. In the filtering step, a spatial index is built over all the objects to help filter out as many objects as possible. In the refinement step, the objects which truly lie within the contact distance of each object are finally determined. In the contact tracing scenario, as the locations of all the objects are constantly changing, the index needs also to be constantly updated with the latest locations of the objects.

The spatial indexes can be categorized as data-partitioning indexes, of which one leaf node is an object (e.g. R-tree [15]), or...
space-partitioning indexes (e.g. Q-tree [35]), of which one leaf node is a space region that covers one or multiple objects. In GLINT, we choose the space-partitioning index for three reasons. First, the storage cost of data-partitioning indexes to maintain a leaf node for each object, which is a point in the contact tracing scenario, is even larger than maintaining the object itself. Second, space-partitioning index offers the flexibility of adjusting the granularity of the indexing. Third, space-partitioning index makes dynamical index maintenance possible as will be discussed next. Many partitioning methods like fixed-grid [36], Q-tree [35], K-D tree [4], and sortile [42] are introduced to direct the space partitioning. Fixed-grid based methods need least effort for indexing and filtering but only works for the cases that the objects are evenly distributed in the entire space [36, 38, 43], while hierarchical structured tree-based methods work well for general cases. In this paper, we choose to build our method based on Q-tree, it can be easily extended to other hierarchically structured indexes.

4.2 GPU-compatible Partitioning Schema

As dynamic memory allocation is expensive on GPU, all the memory spaces used by the partitioning schema should be pre-allocated with considerations of corner cases. Fig. 3 shows the data structures in the GPU memory which are used to maintain a Q-tree structured partitioning schema for indexing the objects. One tree node list node_lst is pre-allocated with enough entries to hold all the tree nodes. The first entry always stores the root node of the tree. Each non-leaf node saves the addresses of its children which construct the tree structure. Another data structure part_lst is pre-allocated and divided into fixed-sized partition buffers. Each leaf node of the tree is assigned with one of those partition buffers to hold the IDs of the objects it covers. Besides, to make sure that the partitioning schema can be dynamically maintained, two stacks node_stk and part_stk are used to hold the addresses of the tree nodes and partition buffers that are available.

**Partitioning granularity**: The partitioning granularity, which defines when a node can be partitioned, is bounded by two factors: the minimum physical size of a partition and the maximum number of objects one partition can cover. In contact tracing scenarios, all the objects in a partition are guaranteed to have been in contact with each other if the diagonal length of the partition is smaller than the contact distance. Therefore, a partition should not be further split when its diagonal length is smaller than the contact distance, and the partitioning granularity should be solely determined by the maximum number of objects a partition can be assigned before it needs to be split. The partitioning granularity (part_gran) is a critical factor that can impact the overall processing performance. On one hand, when each partition is too big, the efficiency of the index degrades and more geometric computations are required in the refinement step to evaluate all the objects in one partition. On the other hand, when each partition is too small, more memory space is needed to store both the partitioning schema and the intermediate index retrieval results, and more time is used on partitioning the objects and filtering with the index. We discuss how a proper partitioning granularity can be chosen with experiments in Section 5.4.1.

**Buffer overflow handling**: A partition buffer has fixed capacity. When a partition becomes too small and can’t be further split, with existing partitioning schema, it can be expected that the number of objects covered by some of the leaf nodes can be larger than the partition buffer capacity. To handle that overflow issue, we set the partition buffer capacity slightly larger than the partition granularity when memory space is sufficient. For the cases that
the number of objects assigned to a partition is even larger than the increased capacity, the overflowed objects will be pushed to a dedicated overflow buffer. During query processing, when one object needs to check with an overflowed partition buffer, it needs also check all the objects in the overflow buffer. As discussed in Section 5.3, the overflow buffer needs only be used in extreme cases in order to ensure the correctness of the contact tracing with a minimal impact on the overall processing performance.

4.3 Filtering-aware Object Partitioning

Algorithm 1: Kernel function for partitioning

1. **Function partition:**
   1. **Data:** points_lst, nodes_lst, filter_lst, refine_lst
   2. **Result:** partition objects into proper leaf nodes
   3. `pid ← thread_id();`
   4. `nd ← nodes_lst[0];`
   5. `filter_root ← nodes_lst[0];`
   6. **while** `nd is not a leaf**
   7. `d ← distance to the border of nd;`
   8. **if** `d > contact_dist then`
   9. `filter_root ← nd;`
   10. `nd ← the child of nd that contains points_lst[pid];`
   11. `push(nd, partition, pid);`
   12. `unroll_push(refine_lst, pid, nd, partition);`
   13. ***/ a range query is needed */`
   14. **if** `filter_root is not a leaf**
   15. `push(filter_lst, pair(pid, filter_root));`

After sampling the latest locations of all the objects, the first step is to divide those objects into proper partitions as an indexing process. As listed in Alg. 1, each kernel thread picks one object and assigns it to the target partition following the partitioning schema. Starting from the root node, the child node at lower levels that covers the object is retrieved until the leaf node is reached (steps 5-9). Then the ID of the object is pushed into the partition buffer that the leaf node is associated with (step 10). In addition, a pair of the target object and the retrieved partition is pushed to a globally shared list refine_lst as the input of the refinement step (step 11). Note that if the distance between the target object to the boundary of the retrieved partition is larger than the contact distance, it is guaranteed that all the objects the target object contacts must be in the same partition it lies in. Otherwise, the object is a boundary object and the partitioning schema needs to be retrieved with a range query to locate all the partitions that may contain contacted objects.

Instead of conducting a range-based index retrieval for all boundary objects starting from the root node, additional information can be collected while partitioning to help prune branches for the filtering step. While traversing the tree nodes, the last node for which the distance from the target object to its boundary is larger than the contact distance will be recorded as filter_root (steps 6-8). It can be guaranteed that the contacted objects must be covered by the descendant partitions of filter_root, and the filtering step can start from filter_root directly instead of the root. A pair of the object ID and filter_root is pushed to filter_lst as the input of the filtering step (step 13).

4.4 Filtering with Block-shared Stack

Algorithm 2: Kernel function for filtering

1. **Function filter:**
   1. **Data:** points_lst, filter_lst, refine_lst, filter_stk
   2. **Result:** conduct index look up over boundary objects
   3. `block_stk ← get_block_stack(filter_lst, block_id());`
   4. `sync_push(block_stk, filter_lst[thread_id()]);`
   5. **while** `block_stk ≠ ∅**
   6. `(pid, nd) ← sync_pop(block_stk);`
   7. **for** each child `c` of `nd**
   8. `if` `dist < contact_dist then`
   9. ```
   10. `synchronous statement to avoid competence between threads.
   11. `if` `c is leaf` and `dist ≠ 0` then
   12. ```

For each within tree retrieval, which is a range query, a depth first search (DFS) or breadth first search (BFS) is needed to retrieve all the leaf nodes that are within the range. A DFS, for example, needs to be implemented with recursive calls, or facilitated with a stack. Conducting range query over tree-like indexes is well studied [21, 46], and we adapt the shared-stack method introduced in [21]. Instead of utilizing a single stack to conduct one range query, all the threads of the same block share the same stack. With a block-shared stack, even though the maximum stack size required by each individual query varies, the maximum stack size required by a collection of queries is relatively the same across blocks, thus less extra stack space needs to be allocated to avoid overflow. Furthermore, even though the initial depth of the filtering root for different boundary objects varies, no thread needs to dry run and wait for others to stop as the range queries for up to 1,024 boundary objects are conducted by 1,024 threads in the same block altogether.

As listed in Alg. 2, initially the stack which is shared by all the threads in the same block is retrieved (step 2). Then every thread pushes the entry it is assigned from filter_lst to the stack as the seed and enters the while loop (steps 3-4). Each push and pop operation conducted on the block-shared stack should be followed by a synchronization statement to avoid competence between threads.

Any time a candidate leaf node is encountered, a pair of the target object and the partition buffer of the leaf node will be pushed into refine_lst for further processing in the refinement step (step 12). Note that a pair of the target object and the leaf node that contains it has already been pushed in refine_lst in the partitioning step and will be ignored here (step 11).

4.5 GPU-optimized Contacted Objects Refining

After partitioning objects and filtering with the index, each object is associated with a list of partitions that may contain objects that lie within the contact distance. The distances between those objects and the target object need to be calculated to finally determine
the contacted objects. Alg. 3 shows the kernel function used to conduct refinements over the object-partition pairs retrieved in the partitioning and filtering steps for a single kernel thread.

**GPU-optimized distance calculation:** One naive method to conduct the refining in parallel using GPU is to use one thread to calculate the distances between one object and the objects in one partition. However, due to the nature of the space-partitioning index, the number of objects in each partition may not be a constant and varies from 0 to the partition buffer capacity. Due to the limitation of the GPU parallelism (Section 2), for the threads in the same warp, the refinement latency is bounded by the object-partition pair that contains the maximum number of objects in the partition. To further improve the efficiency of distance calculations with GPU, we unroll the process of each object-partition pair evaluation into multiple sub-routines, each of which processes a fixed number of object-object pairs, or the refinement granularity (refine_gran). For instance, when the partition capacity is 100, and we set the refinement granularity as 10, then processing one object-partition pair with 66 objects in the partition will be implemented with 7 sub-routines, of which 6 with 10-object-object evaluations and one with 6 object-object evaluations. Each of those sub-routines is then processed with one single GPU thread. As a result, less GPU cycles are wasted during the execution, and the overall refining performance is improved. However, additional execution time is needed to generate sub-routines and additional memory space is required to store the configurations of those sub-routines. That tradeoff will be evaluated with experiments in Section 5.4.2.

**Duplicate computation avoidance:** Euclidean distance is reflexive, such that the distance between two objects will be calculated twice. To avoid duplicate calculation, in the filtering step, the candidate partitions that are situated at the left and bottom of the target objects will also be filtered out. In the refinement step, when calculating the distances of the objects in the same partition of the target object, only the objects whose ID is larger than the target object will be checked. We omit such statements in the algorithms to improve the conciseness of the paper.

### 4.6 Spatial-Temporal Contact Evaluation

**Algorithm 4: Kernel function for evaluating contacts**

```plaintext
1 Function evaluate_contacts:
   Data : contact_tbl, cur_time, contact_time
   Result: evaluate contacts in the hash table
   contact ← contact_tbl[thread_id()];
   if contact ≠ ∅ and contact.end < cur_time then
      if contact.start − contact.end ≥ contact_time then
         push(contact_lst, contact);
      delete(contact_tbl, contact);
```

As discussed in Section 2, a contact between two individuals may cause an infection only when it lasts for a long enough period of time (contact_time), and the contacts which are too short should be pruned. Therefore, all the retrieved pairs of contacted objects in the refinement step need to be evaluated first before being persisted as a valid contact. As each contact is a Key-Value pair, where the key is the IDs of the contacted objects, and the value is the features describing the contact like the start and end time points of the contact, a hash table (contact_tbl) can be used as a working bench to spatially and temporally evaluate the contacts. During the refinement step (Alg. 3), whenever a contacted pair of objects is identified, contact_tbl is queried with the key generated with the IDs of those two objects (steps 7-8). When there is an entry with the same key in contact_tbl, it means that the contact is started in the past, and the end time will be updated to the latest (steps 9-10). Otherwise, the contact is just started and will be inserted into contact_tbl (steps 12-13). After all the active contacts are retrieved with the latest locations of objects, all the entries in contact_tbl need to be evaluated to remove the contacts that are completed. As listed in Alg. 4, each kernel thread evaluates one entry (step 2). If one contact is completed (step 3) and it is long enough (step 4), it will be pushed to contact_lst for persistence (step 5). Otherwise, it will be pruned. The completed contacts are then deleted from contact_tbl (step 6).

**Hash function:** A critical factor to implement an efficient hash table is to evenly distribute the Key-Value pairs into buckets to avoid data skew. One fact in contact tracing scenario is that the number of objects a specific object contacts at a certain time point may vary, but when the object IDs are arbitrarily assigned, which is assumed in this article, the combination of the IDs of two contacted objects are evenly distributed in the space. Therefore, we map the IDs of two contacted objects to one single integer as the key with the Cantor pairing function (Alg. 3 step 7) [20]. As a result, the contacts are evenly assigned to all buckets which leads to a better hash performance.

### 4.7 Dynamic Partitioning Schema Maintenance

As the objects are constantly moving, the partitioning schema needs also to be constantly updated to catch up with the changes in object distribution. Note that partitioning the space referring tens of millions of objects from scratch is time consuming on GPU as significant number of conditional statements to achieve node split and merge [19, 21, 31, 46]. However, in the contact tracing scenario,
Algorithm 5: Kernel functions for updating partitioning schema

<table>
<thead>
<tr>
<th>Function split_node:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data : node_lst, node_stk, part_stk</td>
</tr>
<tr>
<td>Result: split the overflow nodes</td>
</tr>
<tr>
<td>node ← node_lst[thread_id()];</td>
</tr>
<tr>
<td>if node is leaf and node.size &gt; part.size then</td>
</tr>
<tr>
<td>split(node);</td>
</tr>
<tr>
<td>for each child cnode of node do</td>
</tr>
<tr>
<td>cnode ← pop(node_stk);</td>
</tr>
<tr>
<td>if node.partition ≠ ∅ then</td>
</tr>
<tr>
<td>node.partition ← node.partition;</td>
</tr>
<tr>
<td>node.partition ← ∅;</td>
</tr>
<tr>
<td>else</td>
</tr>
<tr>
<td>node.partition ← pop(part_stk);</td>
</tr>
<tr>
<td>push(part_stk, cnodes);</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Function merge_node:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data : node_lst, node_stk, part_stk</td>
</tr>
<tr>
<td>Result: merge the deficient nodes</td>
</tr>
<tr>
<td>node ← node_lst[thread_id()];</td>
</tr>
<tr>
<td>if node is not leaf and node.size &lt; part.size then</td>
</tr>
<tr>
<td>for each child cnode of node do</td>
</tr>
<tr>
<td>if node is not leaf then</td>
</tr>
<tr>
<td>Return;</td>
</tr>
<tr>
<td>for each child cnode of node do</td>
</tr>
<tr>
<td>if node.partition = ∅ then</td>
</tr>
<tr>
<td>node.partition ← cnodes.partition;</td>
</tr>
<tr>
<td>else</td>
</tr>
<tr>
<td>push(part_stk, cnodes.partition);</td>
</tr>
<tr>
<td>push(node_stk, cnodes);</td>
</tr>
</tbody>
</table>

given the fact that the sampling interval is short, the distributions of the objects at two continuous time points will not change significantly. Therefore, it is unnecessary to always re-partition the entire space with the latest object locations. On the other hand, the distribution of object could change when the time scale stretches longer. For instance, people may concentrate in the downtown area in the day time but then move to the urban area after work.

Considering all those facts, instead of reconstructing the partitioning schema after each round of processing, we update the schema by making adjustment at the leaf node level. After processing one batch of locations, the number of objects in each partition is known. The leaf nodes whose partition buffers are overflowed will be split, and the nodes whose children are all leaf nodes and the total number of objects they cover is smaller than the partition capacity will be merged. To avoid using locks on GPU, the schema update process is implemented with two steps: a split step and a merge step, and each node will be evaluated twice (Alg. 5). Note that the reclaimed partition buffer of the parent node will be reused by one of its children (step 8) and vice versa (step 20), thus the node stack (node_stk) and partition stack (part_stk) need only to be popped in the split step and pushed in the merge step, thus no lock is needed. With the help of the massive parallelism of GPU, the schema update process can be completed extremely fast compared with building the partitioning schema from scratch (at sub-millisecond level), and achieves comparable indexing efficiency as proven with experiments in Section 5.3.

5 EXPERIMENTAL EVALUATION

In this section, we evaluate the efficiency of GLINT and show how it can be tuned to achieve optimized performance.

5.1 Setup

We run our tests on a workstation with a 24-core CPU (AMD Ryzen Threadripper 3960X at 3.8GHz) which supports 48 threads in maximum and a GPU (NVIDIA GeForce RTX 2080 Ti with 11GB memory and 4,352 cores). It comes with 128GB memory (DDR4 3200) and 2TB SSD (NVMe M.2 PCI-Express 3.0). The OS is Ubuntu 20.04.4 with 5.4.0-40 kernel and the CUDA version is 11.2.

5.2 Synthetic Data

Many simulators are implemented to generate traces of objects. For example, GMSF [3], short for Generic Mobility Simulation Framework, generates trajectories of objects randomly (GMSSF) or following a Manhattan pattern (GMSFM). The GIS mode of GMSF and another simulator SMARTS [44], short for Scalable Microscopic Adaptive Road Traffic Simulator, generate trajectories of vehicles following the real-world driving rules and maps. However, they all assume that the objects are moving along roads. To better simulate the movement of real-world people, we implemented a Real-world People Movement Simulator (short as RPMS). In RPMS, a person can be in a state of static, walking, or driving. A static person will stay at a certain location for a random period of time, a walking person will walk following roads or in blocks between roads in low speed, and a driving person will drive between a source and destination following the road network in high speed. The initial locations of people are generated following the distribution of coordinates collected from geo-tagged tweets. The source and destination of each driving route referring to the taxi trips data released by the Chicago Data Portal [30]. We generated 10 Million traces fit in a squared space in a size of the metro area of Chicago (40KM×40KM), and each trace samples the locations of one object every second for 1,000 seconds.

5.3 Efficiency of GLINT

We implemented all the steps listed in Fig. 2 with both GPU and CPU for comparison. Each kernel function is executed by one GPU kernel thread or one CPU thread. The maximum contact distance and minimum contact duration are set to 2 meters and 10 seconds respectively. Fig. 4a shows the average execution latency for processing one batch of sampled locations with GPU and CPU. They show that using GPU achieves 5X and 138X speedup over using CPU with 48 and 1 threads respectively (Fig. 4b). As shown in Fig. 4c, the majority of the 222 ms processing time is taken by the partitioning step (15.02%), filtering step (11.7%), and refinement step (66.7%). In comparison, the schema update step and contact evaluation step can be completed extremely fast using GPU (around 1 ms). In addition, 836 MB extra GPU memory spaces are utilized to facilitate the computations on GPU. Fig. 4d shows the portions each in-memory data structure takes. It shows that the memory space
are majorly used to store the IDs of objects assigned to each partition (part_lst 16.27%), the object-partition pairs for refinement step (refine_lst 29.43%), and the active contacts (contact_tbl 45.45%).

Partitioning schema construction: Next we evaluate the effectiveness of updating the partitioning schema at leaf node level. We implement the partitioning schema construction on GPU with a bottom-up method [19, 46]. Initially the space is virtually partitioned into small cells whose diagonal sizes are smaller than the contact distance. All the objects are then assigned to the cells that cover them and the number of objects covered by one cell is recorded by one counter. Finally bottom-up node merges are conducted to construct the tree structure. We conduct three different tests to construct the partitioning schema at each round of processing by 1. re-partitioning the entire space with the latest locations (reconstruct), 2. updating the schema at the leaf node level (update), and 3. reusing the same schema (static). Fig. 5a shows the percentages of the overflowed partitions and the standard deviation of partition sizes as the time goes on for all the tests. It shows that constantly updating the partitioning schema at the leaf node level could make the partitioning schema follow the changes of the object distribution and achieve a fair object partitioning. Note that the majority of the overflowed partitions can be handled by slightly amplifying the partition capacity. Fig. 5b shows the latency of the refinement and schema construction steps for all tests. It shows that updating the schema at the leaf node level achieves almost identical refinement performance as rebuilding the schema from scratch but with far less overhead.

5.4 System Tuning
Next, we show how the configurable parameters can be tuned to achieve the optimized processing performance.

5.4.1 Partitioning granularity. The partitioning granularity, or the maximum number of objects a partition holds before it needs to be split, impacts the the partitioning, filtering, refinement, and the schema update steps. As shown in Fig. 6a, a smaller granularity brings less geometric computations and the refinement latency decreases linearly as partitioning granularity decreases. However, smaller granularity also indicates that the space is partitioned into more partitions, and more time is spent on partitioning objects and filtering with the index. Furthermore, more partitions also brings more load on updating the partitioning schema. Therefore, the overall processing latency is not monotonically decreasing as the partitioning granularity decreases. In addition, smaller partitioning granularity also needs more memory spaces to maintain almost all of the data structures (Fig. 6b). To sum it up, it is unnecessarily saying that the smaller partitioning granularity the better. In this paper, we set the partitioning granularity to 50 as the default value if not mentioned explicitly.

5.4.2 Refinement granularity. The refinement granularity, or the maximum number of object-object pairs one kernel thread needs
to process in the refinement step could impact the performance of the refinement step. Fig. 6c shows the latency of the refinement step and the memory used to store refine_lst for the tests with varying refinement granularities. It shows that the latency does not always decrease with a smaller refinement granularity. That is because additional calculations and memory accesses are needed to unroll the object-partition pairs. When the refinement granularity is too small, the time spent on unrolling could be even larger than the time saved for conducting geometric computations with smaller granularities. Furthermore, the memory space needed to store refine_lst is inversely proportional to the refinement granularity. Overall, the refinement unrolling could further reduce the refinement latency by up to 25%. The refinement granularity is set to 20 by default for all tests.

5.4.3 Bucket number. The hash table maintains the contacts that are active in the two latest time points and the bucket number is configurable. The bucket number impacts the refinement step as an update-or-insert operation is needed when each contacted pair of objects is determined, and the contact evaluation step as all the entries in the table are evaluated. As shown in Fig. 6d, the refinement latency decreases slightly with the increase of the bucket number as less conflicts will be encountered during the insertion. Meanwhile, the time used to evaluate the entries in the hash table increases linearly. In total, the overall processing latency changes only slightly with a larger hash table. Therefore, we choose the size of the hash table referring the number of active contacts, and guarantee the hash table at most 80% full in all tests.

5.5 Varying Contact Tracing Scenarios

The maximum contact distance, minimum contact duration, data distribution, data size, and the GPU used are different for different contact tracing scenarios. We further study how GLINT performs when contact tracing scenarios vary.

5.5.1 Maximum contact distance. A larger contact distance means that more candidate partitions can be retrieved in the filtering step, more geometric computations are conducted in the refinement step, and more active contacts need to be maintained in the hash table. Fig. 7a and 7b show that the average time and memory used to process one batch of locations increase with the increase of the contact distance. The average number of objects that lie within the contact distance for each object is also given for a reference.

5.5.2 Minimum contact duration. The minimum duration of a valid contact impacts the time spent on contacts evaluation and the total number of valid contacts that need to be persisted. As presented in Fig. 7c, the number of valid contacts retrieved each second decreases as the minimum duration of a valid contact increases and more contacts are pruned. As each of those contacts need to be pushed to the contact list (Alg. 4 step 5), the latency to evaluate the entries in the hash table also slightly decreases. However, as the hash table evaluation can be completed fast with GPU, the overall latency decrease is negligible. As a whole, the minimum contact duration only impacts the memory and disk space to store those valid contacts.

5.5.3 Data distribution. We also conducted tests with datasets generated using other simulators. Fig. 7d shows the average number of contacts each object has and the average processing latency for all tests. It indicates that the processing latency is highly related to the distribution of the objects. For the traces generated with the SMARTS simulator, all the data are concentrated along the roads which makes each object on average has 37.4 objects lie within 2 meters from it which only happens in extreme cases.

5.5.4 Data size. To push the capability of GLINT into limits, we extended the number of the objects in the dataset up to 50 million. In addition, two datasets are generated for each data size: one with fixed area and another with fixed density. As presented in Fig. 7e, the overall processing latency increases almost linearly when the data size increases with a fixed density, and the processing latency is higher for dataset with denser populations.

5.5.5 GPU types. Different types of GPUs may be available in different application scenarios. Besides the NVIDIA RTX 2080 Ti graphics card we used for all the tests mentioned above, we also conduct the test on one NVIDIA Tesla K80 graphics card which is relatively old and low in performance, and NVIDIA Tesla V100 graphics card which is an up to date high end GPU. Fig. 7f presents the maximum throughput different GPUs can handle, it shows that tens of millions scale sub-second level real-time contact tracing data processing can be achieved with even the K80 GPU which makes GLINT applicable to broader application scenarios.

Figure 7: The efficiency of GLINT on supporting varying contact tracing scenarios
6 CONCLUSION

In this paper, we propose GLINT, a GPU-based real-time contact tracing system. GLINT provides filtering aware partitioning with dynamically updated schema for indexing moving subjects, and a tree retrieval method with block-shared stack to conduct range-based retrieval of contacted subjects. The querying method takes advantage of the filtering aware partitioning and makes full use of GPU kernel threads for identifying boundary objects. GLINT conducts refinements over object-partition pairs with sub-routines which significantly reduces wasted GPU cycles. The contacts are spatially-temporally evaluated with a GPU-based hash table, and the valid contacts can be identified with minimized overhead. GLINT achieves sub-second contact tracing for the scale of tens of millions subjects with GPU under various tracing scenarios.

ACKNOWLEDGMENTS

This research is supported in part by grants from National Science Foundation ACI 1443054 and IIS 1541063, and National Institute of Health U01CA249296.

REFERENCES

tional Conference on Advances in Geographic Information Systems. 1–10.
[27] Haoyang Vo, Ablimit Aji, and Fusheng Wang. 2014. SATO: a spatial data par-
tional Conference on Advances in Geographic Information Systems. 1–10.