Parallel computing algorithm for real-time mapping between large-scale networks

Ethan Zhang[1], Amirmahdi Tafreshian[1] and Neda Masoud[1]

***Abstract*— In this paper, we propose a scalable massively-parallel algorithm to solve the general mapping problem in large-scale networks in real-time. The proposed parallel algorithm takes advantage of GPU architecture and launches millions of workers to calculate values on a target network simultaneously. Threads are managed through the SIMT execution model and target values are updated through atomic operations. Our experiments show the proposed algorithm can accomplish network mapping (find importance weights for links in a real-world large-scale shared-mobility network) with more than 2 million weights within 1.82 μs (microsecond-level), which is truly real-time. The algorithm performance suggests that mapping computations may no longer be the bottleneck in highly dynamic network-centered problems, as the computations can be completed faster than the solid state drive (SSD) read access latency. Compared to serial algorithms, the speedup is more than 12,000 times. The proposed algorithm is also scalable. Results on simulated data show that even when the network size grows exponentially, microsecond-level computing performance can still be obtained, and even more than 190,000 times speedup can be achieved. The proposed algorithm can serve as a cornerstone for ultra-fast processing of highly dynamic large-scale networks.**

***Keywords*: Parallel computing, Network mapping, CUDA**

I. INTRODUCTION

Recent advancements in communications technology have created high levels of inter- and intra-network connectivity. This connectivity has increased the rate and volume of data exchange as well as the range of data transmission. Higher levels of connectivity have had transformative effects in many domains, such as social and medical sciences, and are on the verge of revolutionizing others, such as transportation networks. To take advantage of the diverse and high-volume data available in this big-data era, there is a need for developing computational methods that enable real-time decision-making. A state-wide transportation network can include millions of nodes evolving dynamically, with critical geolocation and traffic stream-related information changing in real-time. This makes real-time decision-making with traditional, sequential/serial algorithms in such large-scale networks nearly impossible, as these methods consume tremendous time. Carrying out computations in networks typically requires frequent weight updates on the corresponding graphs. In practice, due to the large scale of the networks, it might be challenging to obtain the node/link weights of the target networks directly from the initial networks. As such, creating inter-mediate layers/networks may help speed-up the process. There are also instances in which weights of a target network are computed directly based on weights from other networks [1]. In both cases, the goal is to obtain values for the elements on the target layer (nodes or arcs) based on the values of the source layer, and perhaps through some intermediate layers. Inspired by the "virtual network mapping" problem in [2], we refer to this type of problems as the *network mapping* problems. The network mapping problem is a general problem that arises in distributed networks and multi-layer networks. In these networks, a network/controller/monitor generally needs to gather values/results from other networks and update its own values based on a set of rules [3].

Real-time analysis and decision-making on large-scale transportation networks are challenging due to the need for frequent processing of large streams of data. Real-time route planning, path-planning, and forecasting are some of the application domains where big data can be used to enhance the quality of decision-making. However, there are not much studies in the literature that address the resulting computational burden. Delayed analysis can lead to performance loss, which can prove costly in safety-critical operations. (e.g., advanced driving assistance systems (ADAS) or Connected Automated Vehicle (CAV) operations). Furthermore, the promise of a revolution in the transportation network that is based on high-frequency real-time data streams can only be materialized if such data can be processed and analyzed in real-time. Therefore, to obtain up-to-date processing results, microsecond-level computation is required. Current methods for calculation on large-scale networks can be categorized into three classes: (i) solving problems with time-delay, i.e., provide delayed analysis results; (ii) reducing computational complexity by solving approximate problems; and (iii) employing large computer clusters (e.g., supercomputers or cloud computing clusters) to enable real-time analysis.

In this work, we propose a scalable parallel algorithm that utilizes the CUDA parallel programming platform [4] to solve mapping problems in large-scale transportation networks in the matter of microseconds. Our proposed algorithm utilizes a single graphics processing unit (GPU) device. The proposed method is truly real-time (faster than solid state drive (SSD) read access latency) and less costly than supercomputers or cloud clusters. In contrast to serial algorithms that use a single worker, the proposed algorithm employs millions of simultaneous workers to compute values on the target network. Compared to other parallel algorithms designed to use the central processing unit (CPU) architecture, the proposed algorithm utilizes a single instruction command

[1]E. Zhang, A. Tafreshian and N. Masoud are with Department of Civil and Environmental Engineering, University of Michigan, Ann Arbor, MI, USA, 48105 shuruiz@umich.edu;atafresh@umich.edu; nmasoud@umich.edu.

to control massively asynchronous threads, and synchronizes them with barriers and atomic operations, thereby greatly reducing the solution time complexity. It also does not require the use of supercomputers or computer clusters to achieve real-time performance, which makes the algorithm much more accessible and implementable for real-time systems.

The proposed parallel algorithm uses the *single instruction multiple threads* (SIMT) execution model to manage parallel threads/workers to compute the values on the target network. For each value in the source network layer, a thread is automatically generated and assigned to manage it. Threads carrying values are routed by a virtual router, and the corresponding values are sent to desired destinations in the target layer to update the target value asynchronously. Since the whole process is massively parallel (i.e., for all target element values, their calculations are started at the same time rather than in sequence), the total computing time is shortened to the computation time of computing a single target element in the target network. That is, assuming the size of the target and source layers to be m and n ($n > m$) respectively, time complexity of the proposed algorithm is close to $O(\frac{n}{m})$. Compared to traditional sequential/serial algorithms, whose time complexity is $O(n)$, the speedup is tremendous considering the large values of m and n in large-scale networks.

II. Related work

Real-time solutions in transportation systems not only allow local and state agencies to allocate resources more effectively, but also enable individual travelers to better plan their future activities. Hence, many efforts have been made to boost the computational efficiency of transportation-related problems. However, most of the methodologies used to increase time efficiency rely on cloud computing platforms or computer clusters [5], [6]. In general, the existing computational methods in the transportation literature either rely on sequential/serial algorithms or CPU-based parallel algorithms. CPU-based algorithms suffer from high implementation cost and much slower speedup as well as lower efficiency. Serial algorithms can be time-consuming and are not scalable. For example, assume we need to use values of a source network layer to calculate target network values and each target network element has its own mapping rule. A serial algorithm reads values from the source network layer and calculates results one by one. Considering a target network with several million elements, the process will be executed millions of times, which can take anywhere from several minutes to several days. In other words, these methods either require huge amounts of computational resources or they cannot produce real-time results.

The problem of mapping between networks has been of interest in multiple domains for a long time. Efforts on addressing the time complexity of mapping have increased only in the recent years, due to the emergence of high-frequency and diverse data sources consumed by real-time operations. Y. Chou et al. use an approximate method to calculate shortest path [7]. To solve the shortest path in a large graph, when aggregating the sub-graph results, a mapping process appears to connect the sub-graph results to original graph. Ines Houidi et al. proposed a distributed algorithm to achieve fast mapping between substrate network and the target virtual network in a highly dynamic and changing environment [2] in the communication field. It maintains up-to-date information by introducing multiple distributed agents and letting agents communicate with each other to calculate the virtual network (target network). With the distributed algorithm, workload can be distributed to several workers. However, the algorithm can have time delay and message overload because of the communication between those distributed agents. Time delay and the number of required messages grow tremendously with the network size, which impairs its performance when applied to large-scale networks.

Parallel computing already shows its potential in transportation networks. M.R. Hribar et al. implement the parallel shortest path algorithm for transportation application [8]. When calculating a 257×257 matrix, its relative speedup is 8 times with 128 CPU cores. Although the speedup and efficiency of their approach are not significant comapred to the current parallel computing methods, it is still much faster than the serial methods. G. Ghiani et al. also indicates parallel computing can provides fast solution on the vehicle routing problem [9]. Previous parallel computing algorithms and methods are mainly based on CPU architecture, which requires a large number of CPU cores to achieve a large speedup. CPU-based parallel computing can either result in a large number of communication messages between workers (CPU cores) or fail to achieve microsecond-level computation. For example, message passing interface (MPI), which is the most popular communication method used to conduct parallel computing on distributed memory CPU cores, can be extremely slow when the communication tunnel is blocked by large amount of upcoming messages. This makes it unlikely to obtain real-time information when the target network is large-scale. With the development of graphics processing units (GPU) technology in the recent decade, massively parallel computing can be achieved in a more efficient way. GPU uses more transistors on data processing while CPU uses lots of transistors on data caching and flow control, which makes GPU more suitable for computationally intensive tasks [10].

In our work, we propose a GPU-based parallel algorithm to address the mapping problem. This algorithm gains more than 12,000 times speedup with only one GPU device, when solving the problem of finding importance weights of all links in a shared mobility network with more than 2 million nodes. We also show that even if the network size grows exponentially, real-time (microsecond-level) performance still holds and the speedup exceeds a factor of 190,000. D. Kirk et al. indicate that 2 or 3 times speedup are *just faster* and more than 100 times faster is *fundamentally different* [11]. Based on the performance of the proposed algorithm, we believe that this methodology can be used to alleviate the computational burden of future mapping problems in the field

of transportation.

III. Problem Formulation

1) Source network layer: In the mapping problem, let us denote χ as the source network layer and set $\mathscr{S}$ as the source network set, i.e., the source network layer (source) can contain multiple source networks. We denote the source network $s_i \in \mathscr{S}$ as a directed graph $G_{s_i} = (N_{s_i}, L_{s_i})$, where N_{s_i} is the node set and L_{s_i} is the edge/link set. Both nodes and links can have values associated with them. To make the problem statement more concise, and without loss of generality, in this paper we assume that only links hold values. For any $s_i, s_j \in \mathscr{S}$ and $i \neq j$, $L_{s_i} \cap L_{s_j} = \emptyset$. The node and link sets in the source network layer are represented as:

$$\mathscr{N} = \bigcup_{i=1}^{k} N_{s_i}, \quad \mathscr{L} = \bigcup_{i=1}^{k} L_{s_i} \tag{1}$$

where k is the size of set $\mathscr{S}$. The source network layer provides data and information to calculate the values on the target network.

2) Target network layer: The target network, denoted by $\mathscr{T}$, utilizes data from the source network and is shared by all source networks. Let us define the directed graph $G_{\mathscr{T}} = (N_{\mathscr{T}}, L_{\mathscr{T}})$ to correspond to the target network (target). The target network can be computed based on the information provided by the source networks.

3) Mapping rules: To calculate values of the target network layer from the source network layer, a set of mapping rules are required. Let us define the set of mapping rules as M. Mapping rules are element-wise; that is, each element e in the target network has its own mapping rule, F_e. An $F_e \in M$ is a mapping whose inputs are a set of elements from the source layer and a set of operations to be performed on them, and its outcome is the value of element e in the target network. We re-formulate the problem so as to have the same number of mapping rules as the size of the source layer. Elements of the source layer that do not contribute to the target value computations will have empty mapping rules associated with them.

4) Objective: The objective of the network mapping problem is to find $\mathscr{T} = M(\chi)$, i.e., to calculate values of the elements in the target graph based on the source network layer. Hence, the goal is to calculate $L_{T_{ij}} = F_{T_{ij}}(\mathscr{N}, \mathscr{L})$, $\forall i, j \in N_T$, where $L_{T_{ij}}$ denotes link (i, j) in the target network graph, and $F_{T_{ij}}$ is the mapping rule for link $L_{T_{ij}}$.

IV. Parallel Mapping Algorithm

1) parallel architecture: With the aforementioned formulation, the goal is to calculate all $L_{T_{ij}}$ with its $F_{T_{ij}}$. When calculating all $L_{T_{ij}}$, the traditional serial way is comprised of the following steps:

1) for each operation in the reformatted mapping rule, obtain the desired value from the source network layer to prepare for updating the target value.
2) update $L_{T_{ij}}$ according to $F_{T_{ij}}$ with obtained values from the source network layer.

With the reformatted source layer size n and target layer size m, assuming all operations take unit time, the time complexity of the serial approach is $O(n)$, which is slow for the large size n.

Assume the values in source layer and target layer are denoted by S and T. To solve this problem, as each target element is the results of $F_{T_{ij}}$, we launch tons of threads and assign one thread to each value in the source layer automatically according to the size of S . The maximum number of threads of a CUDA-enabled GPU device is $32 \times (2^{32} - 1) \times 65535 \times 65535$ [12]. Therefore, the method is capable of handling almost all large-scale transportation networks known so far. We reshape the source and target into one-dimension stencils (1-D array) and construct the mapping rules as a *bridge* between the source and target stencils. A thread goes to the source layer to get a value and uses the *bridge* as a virtual router. Each element in the target layer serves as a destination for threads. Since each thread carries a value obtained from the source layer and the value is used to update an element in the target layer, the *router* assigns the thread to its destination. Once a thread is routed to its destination, its value is used to update the target element according to its mapping rule. The thread routing architecture is illustrated in Figure 1.

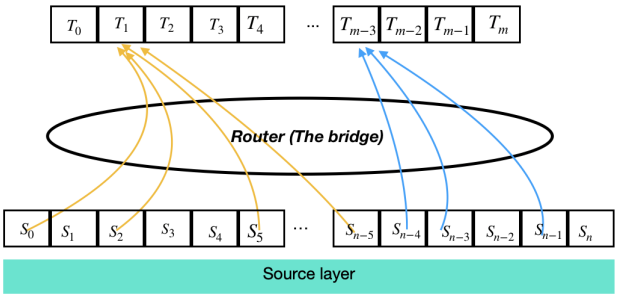

Fig. 1: Threads routing illustration between source layer and target layer

Note that if a value in the source layer is used to calculate multiple target values, or if a target value is used to calculate another target value, dummy elements can be introduced and appended to the source layer (with the corresponding mapping rules added to the *router*) so that each thread only deals with one value of the source stencil. In Figure 1, let the elements of the target layer be the sum of some elements in the source layer. Suppose based on a mapping rule we have $T_1 = S_0 + S_2 + S_5 + S_{n-5}$, as depicted in Figure 1. Thus, the 4 threads that carry S_0, S_2, S_5 and S_{n-5} will be routed to destination T_1. Similarly, another example in Figure 1 shows the mapping rule S_{n-4}, S_{n-3} and S_{n-1} will be routed to T_{m-3}. All threads work in parallel. Since we are dealing with large-scale networks, the size of the mapping rule set M is tremendous. Managing such large number of workers is a complex task. Therefore, we use the SIMT execution model, which uses a single general execution command

to control all threads. Note that in parallel computing, we have to control the behavior of each worker instead of controlling the whole process, i.e., the algorithm is designed to control one worker, yet all workers can use the same algorithm. Unlike CPU-based parallel computing (e.g., MPI based parallel computing), with SIMT, no communication exists among workers. In other words, each thread/worker only knows its own status and conducts its own work without knowing status of other threads. Hence, it is hard for the threads to collaborate with each other to achieve their goal. We will discuss the collaboration issue under this situation in the following sections.

2) Thread management: In CUDA, a GPU device is represented by a *grid*. Inside a grid, blocks are arranged in three dimensions, represented by x, y and z. Threads are stored in blocks in three dimensions. Since only current block coordinates and local (block-level) thread coordinates are known to each thread, thread management is necessary to make sure all threads work properly. Due to the complexity of working with three-dimensional grids and blocks (i.e., the resulting nine-dimensional space), and the possible higher latency that could result from working in higher dimensional spaces, we present the thread management method for two-dimensional grids and blocks. With a source stencil of size n, let the block shape be 32×32, i.e., for each block, both its x and y directions have 32 threads, resulting in a $[\lceil \frac{\sqrt{n}}{32} \rceil, \lceil \frac{\sqrt{n}}{32} \rceil]$ grid shape, i.e., the total number of blocks are $\lceil \frac{\sqrt{n}}{32} \rceil^2$. Therefore, total number of threads that can be used is $32^2 \times \lceil \frac{\sqrt{n}}{32} \rceil^2 \geq n$. To effectively manage the millions of available threads, a global thread index is assigned to each thread to serve as its unique ID. Thread ID is essential for the router to distinguish a worker's identity. In this framework, we require that S_i is routed by thread i. The simplest way of indexing threads is using ordered integer values so that we can directly use the indices for counting the total number of threads. The indexing rules are demonstrated in Figure 2.

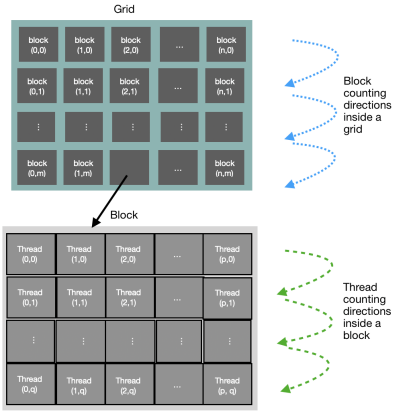

Fig. 2: Threads indexing rule

In both grid and block levels, the following approach is used for indexing: the entity in the northwest corner is assigned the index 0. We index the rest of the entities by first prioritizing the each direction, followed by the south direction. This procedure is demonstrated in Figure 2. Each thread knows the block to which it belongs as well as its own local coordinates within that block, and uses this information for self-indexing (i.e., obtaining a unique global ID). Note that in the CUDA platform, block and thread IDs are counted starting from 0. Assuming a thread t is located in block i, the self-indexing algorithm is demonstrated in Algorithm 1.

Algorithm 1 Thread self-indexing algorithm

Result: Global thread index

Input: Grid dimension $(Grid_x, Grid_y)$, block dimension $(Block_x, Block_y)$

1. Get current block coordinate (b_x, b_y) in the grid, and its own thread coordinates (t_x, t_y) in block i
2. Calculate number of total blocks B and total threads K before i:

$$\begin{aligned} B &= b_x \times Grid_y + b_y + 1 \\ K &= B \times Block_x \times Block_y \end{aligned} \tag{2}$$

3. Calculate global index of thread t:

$$t_{id} = K + t_x \times Block_y + t_y \tag{3}$$

Return t_{id}

Once t_{id} is obtained, the corresponding thread is assigned to element $S[t_{id}]$ of the source layer.

3) Race condition and atomic operation: According to the mapping architecture shown in Figure 1, several threads may need to update one destination value in a mapping process. As we know, all values are stored in the device memory with an address. As we mentioned before, one thread has no knowledge about the status of other threads. That will make several threads update the same address simultaneously, referred to as the race condition [13], i.e., threads are racing to update the same memory and the result of one thread will be replaced/covered by the results of the following thread.

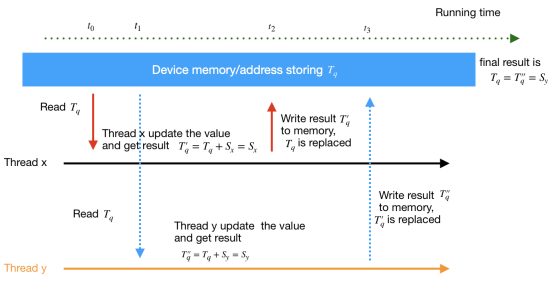

Fig. 3: Race condition between threads in a mapping problem with mapping rule of adding the source values

Suppose we want to calculate the value of element q on the target network based on a simple addition of values of elements x and y in the source network, i.e., $T_q = S_x + S_y$. Recall that we employ a threat for each source-network element, providing us with two threads x and y. This example is illustrated in Figure 3. The work is assigned

to two parallel workers, and each worker needs to do one addition update on T_q. Thread x reads $T_q = 0$ from the device memory at time t_0, and starts performing operations to update T_0 according to the mapping rule $T'_q = T_q + S_x$. At time t_2, thread x completes it computations and writes $T'_q = S_x$ into the memory. Due to the speed difference in command executions between the two threads, thread y reads T_q at $t_1, t_0 < t_1 < t_2$. Since T_q is not yet updated by thread x at time t_1, thread y reads the value $T_q = 0$ from the shared memory. Thread y finishes its task and gets its own calculation result $T''_q = T_q + S_y$ between times t_1 and t_3. At time t_3, it writes the result back into the memory, erasing the results written into the memory by thread x at t_2, as thread y has no knowledge of the status of other threads. Therefore, the race condition results in the final result of $T_q = S_y$, rather than $T_q = S_x + S_y$.

To avoid race condition when solving the mapping problem with millions of parallel threads/workers, we use atomic operation to safely lock the device memory that stores target values and force other racing threads to wait for the lock. In atomic operation, when the target value is read by a thread, the thread will put a lock on the value. The lock is released when the thread updates the target value in memory. While the memory is locked, all coming threads who need to use the locked portion of the memory have to wait for the lock. Once the lock is released, other threads can get the lock and start the process again. Note that the lock on a target memory does not affects threads that operate on other target memories. The process is shown in Figure 4.

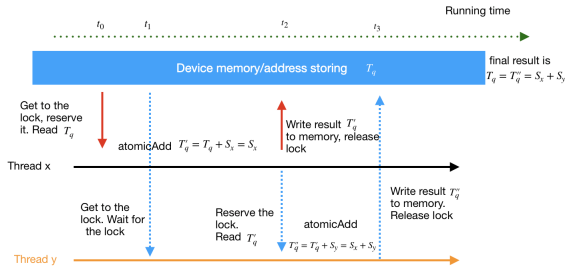

Fig. 4: Atomic operation to get target value correctly

In Figure 4, thread x locks the memory associated with $T(q)$ at time t_0, and starts its computations. At time t_1, thread y attempts to obtain $T(q)$, but since there is a lock on $T(q)$, y can only reserve the lock. At time t_2, x updates the memory and returns the lock, at which point y instantaneously obtains the lock as well as the current value of $T(q) = S_x$. Thread y completes its computations and returns the final value of $T(q) = S_x + S_y$ along with the lock at time t_3. Following this approach, although threads are asynchronous and have no knowledge about status of other threads, they can collaborate with each other to get the correct final values. The proposed parallel algorithm is summarized in Algorithm 2.

V. Numerical Results

In this section, we provide numerical results to show the performance of the proposed algorithm with both real-world and simulated cases. In shared mobility services, a common practice for dealing with stochastic ride-sourcing demand is to route drivers proactively. In order to find high-quality routes, one can use historical data to find the importance (positive weight) of different links of a network discretized in time and space. Then, a useful route can be easily obtained by finding the longest path in the aforementioned network. That is, we aim to search for the route that provides the highest weight (e.g., highest revenue). Given a large-scale transportation network, finding high-quality routes in a timely manner using this procedure can be challenging. Here we use the proposed parallel algorithm to enhance the computational efficiency of this procedure.

Algorithm 2 GPU-based SIMT Parallel mapping algorithm

Result: calculated target network

Input: Source networks $\mathscr{S}$, target network $\mathscr{T}$, mapping rules $\mathscr{M}$

1.Construct the 1-D stencil source layer S, target layer T and mapping rules M according to $\mathscr{S}$, $\mathscr{T}$ and $\mathscr{M}$. Allocate memory for S, T and M on CPU(host) and GPU(device). Copy values from the host memory to the device memory.

2. Construct block shape as 32×32 and grid shape as $\left[\lceil \frac{\sqrt{n}}{32} \rceil, \lceil \frac{\sqrt{n}}{32} \rceil\right]$ grids, where n is the size of source layer stencil.

3. Locate current thread coordinates (t_x, t_y) and block coordinates (b_x, b_y). Calculate current global thread ID t_{id} based on $Algorithm$ 1.

4. Assign thread t_{id} to the source element $S[t_{id}]$. Use the mapping rule to find the destination in the target layer stencil.

5. Once thread t_{id} is routed by the mapping rule and reaches the target address, use $S[t_{id}]$ to perform $Atomic$ operations to update the target address.

6. Synchronize threads in a block to make sure threads in the same block complete their tasks. Exit device and synchronize device on host to wait for threads in different blocks finishing their tasks so that all target elements are updated.

7. Copy the result from device back to host. Output calculated target network $\mathscr{T}$.

Suppose we have a timed origin-destination table that provides the number of customers who started their trips at time t' from station i' to station j'. Further, let a link in the transportation network be presented by a tuple (t, i, j). Given stationary traffic conditions, we can specify whether link (t, i, j) is a part of shortest path for every trip (t', i', j') of customers. Let A be an ordered 1-D array that contains the weight of all valid combinations of (t, i, j) and (t', i', j'). This weight can be any function of number of customers and trip and link costs (e.g., travel times). Suppose that the weight of every link (t, i, j) is the sum of the trip weights in A that corresponds to this link, which shows the importance of the link (t, i, j) (a higher weight indicates more (t', i', j') trips can use the link). Let C be a 1-D array of all link

weights. Finally, let B be a 1-D array of the same size as A that contains the indices of C. Now, we are interested in parallelizing the calculation of the elements of the target array C, based on the elements of source array A and the mapping rules of B. The problem can be posed as $W_{(i,j,t)} = \sum_{(i',j',t')}^{P} W_{(i',j',t')}, \forall (i,j,t)$, if (i,j,t) is contained in trip (i',j',t'), where P is the table that contains all (i',j',t') information. The trips in this case study are obtained from the New York taxi dataset.

With the proposed parallel algorithm, the computing time of the above problem is 0.001824 ms (1.82 μs). The speed is much faster than SSD read access latency (50 μs [14]). For the same problem, the running time of the conventional, serial algorithm is 0.0217 seconds, i.e., the speedup is 1.2×10^4 times compared to serial algorithm. The improvement is tremendous. As future networks are going to be much larger, we expand the scale of the dataset by simulating the source layer values and mapping rules. Based on Amdahl's law, with n parallel workers, the speedup is no greater than a factor of n, i.e., with serial time we can compute the upper bound performance of any CPU-based parallel algorithm. Therefore, only serial CPU time is provided here. The corresponding performances are shown in TABLE I. Running time for serial and the proposed parallel algorithm are recorded in seconds and microseconds separately.

TABLE I: Algorithm Performance

Source size n	Target size m	n/m	Serial time (s)	Parallel (μs)	Speedup
1×10^6	1×10^4	1×10^2	0.004025	14.624	275
2×10^6	1×10^4	2×10^2	0.005206	15.072	345
4×10^6	1×10^4	4×10^2	0.010870	17.088	636
8×10^6	1×10^4	8×10^2	0.019802	17.056	1161
1×10^7	1×10^4	1×10^3	0.026285	17.184	1530
2×10^7	1×10^4	2×10^3	0.050087	15.360	3261
4×10^7	1×10^4	4×10^3	0.104406	14.400	7250
8×10^7	1×10^4	8×10^3	0.222599	241.344	922
1×10^7	1×10^5	1×10^2	0.049775	1.824	27289
2×10^7	1×10^5	2×10^2	0.091239	1.952	46741
4×10^7	1×10^5	4×10^2	0.165317	2.016	82002
8×10^7	1×10^5	8×10^2	0.347196	26.176	13264
1×10^7	1×10^6	1×10^1	0.143704	1.856	77427
2×10^7	1×10^6	2×10^1	0.357694	1.842	194188
4×10^7	1×10^6	4×10^1	0.754735	1.920	393091
8×10^7	1×10^6	8×10^1	1.580698	21.504	73507
1×10^8	1×10^6	1×10^2	2.044502	47.424	43111

The results indicate that even if the source layer size increases exponentially, the algorithm can still achieve microsecond-level performance, which indicates the scalability of the proposed parallel algorithm. Race condition and waiting time spent on locks account for the observed non-linear speedup; that is, the time complexity of the parallel algorithm does not strictly follow $O(\frac{n}{m})$ because when performing calculations, threads have to read global device memory frequently, which can consume a large portion of time. In addition, the number of locks needed by each thread is not uniformly distributed (i.e., not $\frac{n}{m}$). However, the computing time of the proposed algorithm is highly correlated with $\frac{n}{m}$, which is the number of average locks for which a thread has to wait. As shown in the results, when the source layer size is even ten times larger, its computing time can be much shorter if it has a smaller $\frac{n}{m}$ value.

VI. CONCLUSIONS

In this paper, we propose a scalable parallel computing algorithm to solve mapping problems between large-scale networks in truly real-time. We provide a three-layer parallel architecture, which contains a source layer, a router, and a target layer. The algorithm utilizes the great potential of GPU device to generate tons of workers to compute results in parallel. All threads are managed with the SIMT execution model. Collaborations between workers are achieved through atomic operation, which reduce the time complexity of the network mapping problem from $O(n)$ to approximately $O(\frac{n}{m})$, where n, m indicate the source and target layer size, with $n > m$. Our numerical results show that the algorithm can find importance weights of all links in a real-world large-scale shared-mobility network with more than 2 million edges/weights within 1.82 μs (microsecond-level), which is a 12,000 times speedup compared to serial computing. Experiments show that even when the network size grows exponentially, microsecond-level performance can still be obtained, and more than 190,000 times speedup is achieved.

REFERENCES

[1] Y. Zhang, L. Wang, W. Sun, R. C. Green II, and M. Alam, "Distributed intrusion detection system in a multi-layer network architecture of smart grids," *IEEE Transactions on Smart Grid*, vol. 2, no. 4, pp. 796–808, 2011.

[2] I. Houidi, W. Louati, and D. Zeghlache, "A distributed virtual network mapping algorithm," in *Communications, 2008. ICC'08. IEEE International Conference on*. IEEE, 2008, pp. 5634–5640.

[3] D. Peleg, "Distributed computing," *SIAM Monographs on discrete mathematics and applications*, vol. 5, pp. 1–1, 2000.

[4] J. Sanders and E. Kandrot, *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional, 2010.

[5] Z. Li, C. Chen, and K. Wang, "Cloud computing for agent-based urban transportation systems," *IEEE Intelligent Systems*, vol. 26, no. 1, pp. 73–79, 2011.

[6] A. Biem, E. Bouillet, H. Feng, A. Ranganathan, A. Riabov, O. Verscheure, H. Koutsopoulos, and C. Moran, "Ibm infosphere streams for scalable, real-time, intelligent transportation services," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM, 2010, pp. 1093–1104.

[7] Y.-L. Chou, H. E. Romeijn, and R. L. Smith, "Approximating shortest paths in large-scale networks with an application to intelligent transportation systems," *INFORMS journal on Computing*, vol. 10, no. 2, pp. 163–179, 1998.

[8] M. R. Hribar, V. E. Taylor, and D. E. Boyce, "Implementing parallel shortest path for parallel transportation applications," *Parallel Computing*, vol. 27, no. 12, pp. 1537–1568, 2001.

[9] G. Ghiani, F. Guerriero, G. Laporte, and R. Musmanno, "Real-time vehicle routing: Solution concepts, algorithms and parallel computing strategies," *European Journal of Operational Research*, vol. 151, no. 1, pp. 1–11, 2003.

[10] C. Nvidia, "Toolkit documentation," *NVIDIA CUDA getting started guide for linux*, 2014.

[11] D. Kirk *et al.*, "Nvidia cuda software and gpu parallel computing architecture," in *ISMM*, vol. 7, 2007, pp. 103–104.

[12] C. Nvidia, "Nvidia cuda c programming guide," *Nvidia Corporation*, vol. 120, no. 18, p. 8, 2011.

[13] M. J. Quinn, "Parallel programming," *TMH CSE*, vol. 526, 2003.

[14] P. Specification, "Intel® solid-state drive dc s3500 series," 2015.