Mesoscopic Traffic Simulation on CPU/GPU*

Yan Xu and Gary Tan
School of Computing
National University of Singapore
Singapore 117417
xuyan.nus@gmail.com

Xiaosong Li
School of Computing Engineering
Nanyang Technological University
Singapore 639798
xli15@e.ntu.edu.sg

Xiao Song
School of Automation
Beihang Univ.
Beijing, China
songxiao@buaa.edu.cn

ABSTRACT
Mesoscopic traffic simulation is an important branch of technology to support offline large-scale simulation-based traffic planning and online simulation-based traffic management. One of the major concerns using mesoscopic traffic simulations is the performance, which means the required time to simulate a traffic scenario. At the same time, the GPU has recently been a success, because of its massive performance compared to the CPU. Thus, a critical question is "whether the GPU can be a potential high-performance platform for mesoscopic traffic simulations?" To the best of our knowledge, there is no clear answer in the research area. In this paper, we firstly propose a comprehensive framework to run a traditional time-stepped mesoscopic traffic simulation on CPU/GPU. Then, we design a boundary processing method to guarantee the correctness of running mesoscopic supply traffic simulations on the GPU. Thirdly, the proposed mesoscopic traffic simulation framework is demonstrated to simulate 100,000 vehicles moving on a large-scale grid road network. In this case study, running a mesoscopic supply traffic simulation on a GPU (GeForce GT 650M) gives 11.2 times speedup, compared with running the same supply simulation on a CPU core (Intel E5-2620). In the end, this paper explains the theoretical limitation of running mesoscopic supply traffic simulations on the GPU. In conclusion, regardless of high system complexity, the proposed mesoscopic traffic simulation framework on CPU/GPU provides an innovative and promising solution for high-performance mesoscopic traffic simulations.

Categories and Subject Descriptors
I.6.8 [Types of Simulation: Parallel]; D.1.3 [Concurrent Programming: Parallel Programming]

Keywords
Mesoscopic Traffic Simulation, GPU, Correctness, Scalability

1. INTRODUCTION
Traffic simulation is an appealing solution for traffic planners and engineers to solve Dynamic Traffic Assignment (DTA) problems for offline system planning and online operation management. Technically, any traffic simulation consists of two components: 'demand' and 'supply' [1]. Modeling from the travelers' point of view, the former is to understand how travel decisions are made, such as mode choice, departure time choice and route choice. Modeling from the traffic flow's point of view, the latter is to understand how traffic demand is assigned to available road resources. This paper focuses mainly on the supply part which is often more computational costly because the supply models the traffic flow dynamics with vehicles moving on road networks [2].

To simulate the supply part of an entire city such as Singapore [1] or Beijing [4] in peak hours, when a few hundred thousand to a few million vehicles are on the road, the amount of time it would require a single von Neumann-style serial processor to track and compute the states of such large numbers of vehicles makes traffic simulations nearly infeasible on these architectures. To tackle this performance problem, researchers have made efforts on two main approaches, structurally improve supply framework and enhance performance with parallel computing.

To improve efficiency of the supply framework, mesoscopic traffic simulators, such as DynaMIT [3, 4] and DynaSMART [5], were created to reduce the computational requirement of microscopic traffic simulators. Compared with microscopic traffic simulators, individual vehicle dynamics in mesoscopic traffic simulators are approximated by a speed-density model in the moving part of a link and a queuing model in the queue part of a link [1]. At the same time, in contrast to aggregated macroscopic models [6], vehicles are modeled as agents in mesoscopic traffic simulations to gain the advantage that they are consistent with the detailed demand models of traveler behaviors, such as route choice. For their well designed tradeoff between performance and accuracy, mesoscopic traffic simulators have been widely used to support large-scale simulation-based DTA systems [3-5]. A recent work is ETSF [16], which reduces the theoretical time complexity of the mesoscopic supply simulation, with an assumption that vehicles on the same lane of a segment are moving at the same speed at a time step. ETSF is introduced in Section 2.2. However, mesoscopic traffic simulations are still not efficient enough to satisfy the intensive computational requirement of real-world large-scale DTA applications [1]. Moreover, in most cases, tens or hundreds of runs are required for statistical analysis before any decision making, which make the problem more challenging.

Thus many researchers addressed the problem from a parallel computing perspective. They have used multi-core CPUs or CPU clusters to handle the large computational load [7-9]. Traffic network is typically decomposed into segments that are handled by different processors. Load balancing, inter-processor communication, and synchronization then become important considerations. For optimal use of computing resources, segments must be distributed evenly among processors based on the estimated computation cost of each segment. At the same time, the communication cost between processors should be minimized. For this purpose multi-core parallel algorithms and data structures

SIGSIM-PADS'14, May 18–21, 2014, Denver, Colorado, USA.

* This research was supported by the National Research Foundation Singapore through the Singapore MIT Alliance for Research and Technology's FM IRG research programme (sub contract R-252-001-459-592).

have been developed in [8, 9] and Message Passing Interface (MPI)-based communication among CPU clusters is used in [2]. In summary, traditional parallel computing using either CPU clusters or multi-core processors is the main-stream technology to support large-scale mesoscopic traffic simulations. However, it is still a challenge to run a large-scale mesoscopic traffic simulation on a CPU cluster efficiently. Besides, the cost and complexity of maintenance of such computing resources makes these approaches expensive and sometimes undesirable.

Graphics processing unit (GPU) has recently been a success, because of its massive performance compared to the CPU. While GPUs were primarily meant to do three-dimensional rendering in graphics applications, rapid developments in their architectures have enabled their use in scientific computing [11, 22], computational finance [12], computational biology [12], simulations [17, 19-21] and high performance computing [13]. Moreover, the development of direct computing application protocol interfaces (APIs) such as CUDA (compute unified device architecture) [10] and OpenCL have significantly reduced the programming efforts. However, fundamental differences in GPU and CPU architectures mean that traditional technique of converting serial implementations to parallel using standards such as OpenMP and MPI is inapplicable. Thus, a critical question is "whether the GPU can be a potential high-performance platform for mesoscopic traffic simulations?"

There have been some research work to enhance the solution to traffic simulation on GPUs. Perumalla et al. [14] introduced a method to simulate the vehicle movement on GPU by using a field based model. This model maps the real world road data onto a 2D lattice, with each element in lattice representing the possibility of turning either left/right or up/down. By using this possibility data, the vehicles will be directed from one position in the 2D array to another position. The proposed field based model is similar to the classic Cellular Automata Model. However, the contribution of this work within the global traffic simulation research framework is not clarified. The MATSIM team recently released a research work to implement an event-driven mesoscopic traffic simulation framework on the GPU [15]. The paper introduced two kernel functions (*moveLink* and *moveNode*) to implement the core queue simulation. They also talked about three different implementations of the vehicle array in the GPU memory. Their work is pioneering and obtained a speedup over serial applications between 5.5 and 60 times depending on different data structures and NVIDIA GPU series. However, the paper did not confirm the correctness of running MATSIM on the GPU and also did not explain how to migrate a real-world mesoscopic traffic simulation from the CPU to the GPU. Besides, compared to time-stepped mesoscopic traffic simulation (e.g. DynaMIT [1]), event-driven mesoscopic traffic simulation has limitations for online simulation-based operation management [2].

In this paper, we use both the CPU and the GPU to enhance performance of a traditional time-stepped mesoscopic traffic simulation enabled by Entry Time based Supply Framework (ETSF) described by [16]. This paper has four contributions:

1. This paper proposes a comprehensive framework to run a time-stepped mesoscopic traffic simulation on CPU/GPU, including time management, network and vehicle modeling on the GPU, kernel functions on the GPU and incident modeling on the GPU.

2. This paper introduces an innovative boundary processing method to guarantee the correctness of running mesoscopic supply traffic simulations on the GPU.

3. The proposed mesoscopic traffic simulation framework is demonstrated to simulate 100,000 vehicles moving on a grid road network and the supply traffic simulation on a GPU (GeForce GT 650M) gets 11.2 times speedup, compared with running the same supply simulation on a CPU core (Intel E5-2620).

4. Based on profiling results, we found that the theoretical limitation (or the bottleneck) of running mesoscopic supply simulation on the GPU is the memory access latency. It is also a challenge that should be solved in future.

The paper is organized as follows. In Section 2 the GPU is introduced and a brief overview of ETSF method is provided. Section 3 introduces the mesoscopic traffic simulation framework on CPU/GPU. Section 4 discusses the correctness of running mesoscopic supply traffic simulation on the GPU and proposes a boundary processing method. Section 5 further talks about optimization of data transfer between the CPU and the GPU. In Section 6 we provide comparison results of our benchmarks with the serial implementation and analyze the theoretical limitation. Finally, we list conclusions and future works in Section 7.

2. BACKGROUND

2.1 Combination of CPU and GPU

NVIDIA GPUs can be found in roughly 70 million PCs and notebooks around the world [15]. The key to the success of GPU computing has partly been its massive performance compared to the CPU [18]. Nowadays, there is a performance gap between the GPU and the CPU, when comparing theoretical peak bandwidth and gigaflops performance. The performance gap has its root in the physical restraints and the architecture differences between the two processors. GPUs are designed to gain massive performance to address problems that can be expressed as data parallel computations (i.e., the same program is executed on many data elements in parallel) with high arithmetic intensity (i.e., the ratio of arithmetic operations to memory operations). On the contrary, CPUs are designed to address general-purpose problems, which usually have complex execution logic.

One of the major performance factors of CPUs has traditionally been its steadily increasing frequency [18]. However, in the 2000s, this increase came to an abrupt stop. At the same time, GPUs were growing exponentially in performance due to massive parallelism. Parallelism appears to be a sustainable way of increasing performance and there are many applications that are perfectly suited for GPUs. However, increased parallelism can only increase the performance of the parallel code section, meaning that the serial part will soon become the bottleneck. Thus, the combination of traditional CPU cores and a massively parallel GPU can benefit a large number of applications. It is true for mesoscopic traffic simulations.

However, it is a challenge to migrate traditional mesoscopic traffic simulation from the CPU to CPU/GPU, because the GPU programming is quite different from the CPU programming in many ways. First, the GPU can support thousands of light-weight threads, which requires the target problem to be divided into thousands of parallel sub-problems. Second, the GPU has three memory spaces, listed in decreasing order by speed: registers, the shared memory, and the global memory. GPU programming requests users to decide which memory space to use for each dataset in the GPU memory. Besides, in most cases, the amount of total registers and total shared memory is limited. Third, the GPU and the CPU are currently using different memory spaces. The data communication between the CPU and the GPU is time costly. Fourth, the performance of GPU programming is sensitive to

coalesced global memory access and bank conflicts in shared memory access, which do not exist in CPU programming. Finally, the available APIs on GPUs are still not sufficient.

2.2 Entry Time-based Supply Framework

The time complexity of simulating a lane at a time step in current mesoscopic supply framework [1, 2, 4, 5] is linear to *the number of vehicles on the lane*. To simulate an entire city the computation cost is often practically unacceptable, especially in congested traffic scenarios. To tackle this problem, an approach named 'Entry Time based Supply Framework (ETSF)' is proposed to improve the performance of mesoscopic applications. The main idea is that ETSF updates the '*entry_time_to_pass'* and '*entry_time_to_queue'* parameters of each lane to compute the moving/queuing parts of link/lane vehicles. The key feature of ETSF is to reduce the theoretical time complexity of simulating a lane at a time step to *the number of vehicles passing the lane*. ETSF significantly reduces the time to simulate a congested traffic scenario [16]. In the following paragraphs of this section, we try to explain ETSF approach before presenting GPU based ETSF in the next section.

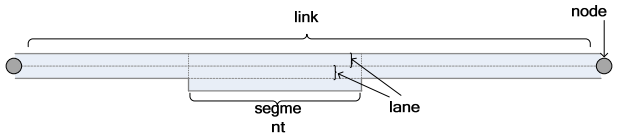


Figure 1. Network-related terminologies used in this paper

As shown in Figure 1, in ETSF, a road network is modeled as *nodes*, *links*, *segments* and *lanes*. The nodes correspond to intersections of the actual road network, while links represent unidirectional path ways between nodes. Each link is divided into a number of segments, according to geometry features. Each segment contains lanes. Each lane contains a number of vehicles which are located on the lane. Each lane has capacity constraints at the upstream end and the downstream end, referred to as the input capacity and the output capacity. A queue occurs in a lane if vehicles cannot pass the lane. A spill-back occurs if a lane is blocked, which means the length of the queue on the lane is equal to the length of the lane.

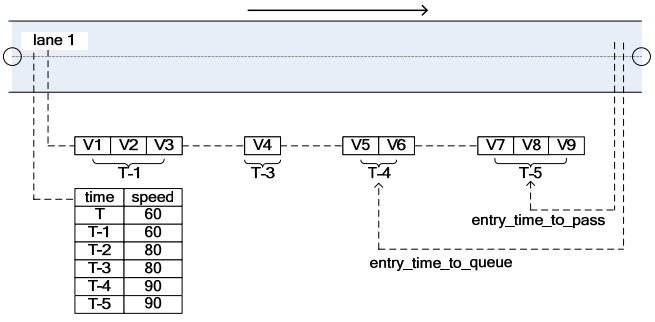


Figure 2. An example of vehicles in a lane of a link in ETSF

Figure 2 shows an example of vehicles in a lane of a link in the ETSF. First, each lane of the link has a list of vehicles which are ordered by their *entry time*, referring to the time a vehicle enters the lane. Second, in ETSF, each lane of the link is devised to have a speed table (see in Figure 2), which contains the lane speeds of recent time steps, with the assumption that vehicles in the same lane are moving using the same speed at a time. Given the speed table and the *entry time* of a vehicle, the accumulated travel distance of the vehicle can be calculated. Third, each lane of the link has a key attribute: *entry_time_to_pass* (t_p), which means that at a time t, if the lane entry time of a vehicle is earlier (or smaller) than (or equal to) t_p, its accumulated movement distance equals to or bigger than the length of the lane and has the potential to pass current lane. For example, in Figure 2 the current time is T and t_p is T-5. Fourth, each lane of the link has another key attribute: *entry_time_to_queue* (t_q), which denotes that at a time T, if the lane has a queue and the entry time of a vehicle is earlier than (or equal to) t_q, the vehicle either enters the queue or passes the lane. For example, in Figure 2, t_q is T-4. It means vehicles whose entry time is earlier than T-4 are either in the queue or have passed the lane. Note that the queue length of a lane is calculated by adding up the occupancy space of vehicles in the queue and t_q also takes into account the vehicles whose accumulated movement distances are smaller than the length of the lane but catch the end of the queue.

3. Mesoscopic Traffic Simulation Framework on CPU/GPU

3.1 The Framework

The major motivation to design a new simulation framework is to make full use of two types of computational resources: central processing unit (CPU) and graphics processing unit (GPU). In this framework, the GPU is responsible for the supply part of mesoscopic traffic simulation, which includes speed calculation, vehicle movement on a road and between roads and queue calculation. A key feature of supply simulation is that the simulation of a road is only related with its surrounding roads, which fits GPU's data parallel requirement. The CPU is responsible for the demand part and the I/O part of mesoscopic traffic simulation, which includes vehicle generation, departure time choice, pre-trip route choice, en-trip route choice and pushing simulation results to files. A key feature of demand simulation is that vehicles are making decisions based on the information on the global road network. Figure 3 shows the mesoscopic traffic simulation framework on CPU/GPU, which explains the logic procedure and the simulation time management. This framework is suitable for general time-stepped mesoscopic traffic simulation [3-5].

The traditional simulation time, which controls the turnover of the system status, is divided into three components: a demand time step (t_d), a supply time step (t_s) and an I/O time step (t_{io}). A traffic simulation is completed only if t_d, t_s and t_{io} are all reaching the simulation end. In this framework, multiple time steps enable to identity the exact progress of different components in a traffic simulation. Note that at an instantaneous time, the three time steps can be different. The time management in this framework is controlled by three rules:

- Rule 1: $t_d >= t_s$
- Rule 2: $t_s >= t_{io}$
- Rule 3: $t_d <= t_s + DPP$

First, t_d is always not smaller than t_s, because only if vehicles entering the simulation at time *t* are generated, the supply simulation at *t* can start. Second, t_s is always not smaller than t_{io}, because only if the supply simulation at *t* is completed, the simulation results at *t* can be outputted to files. The third rule

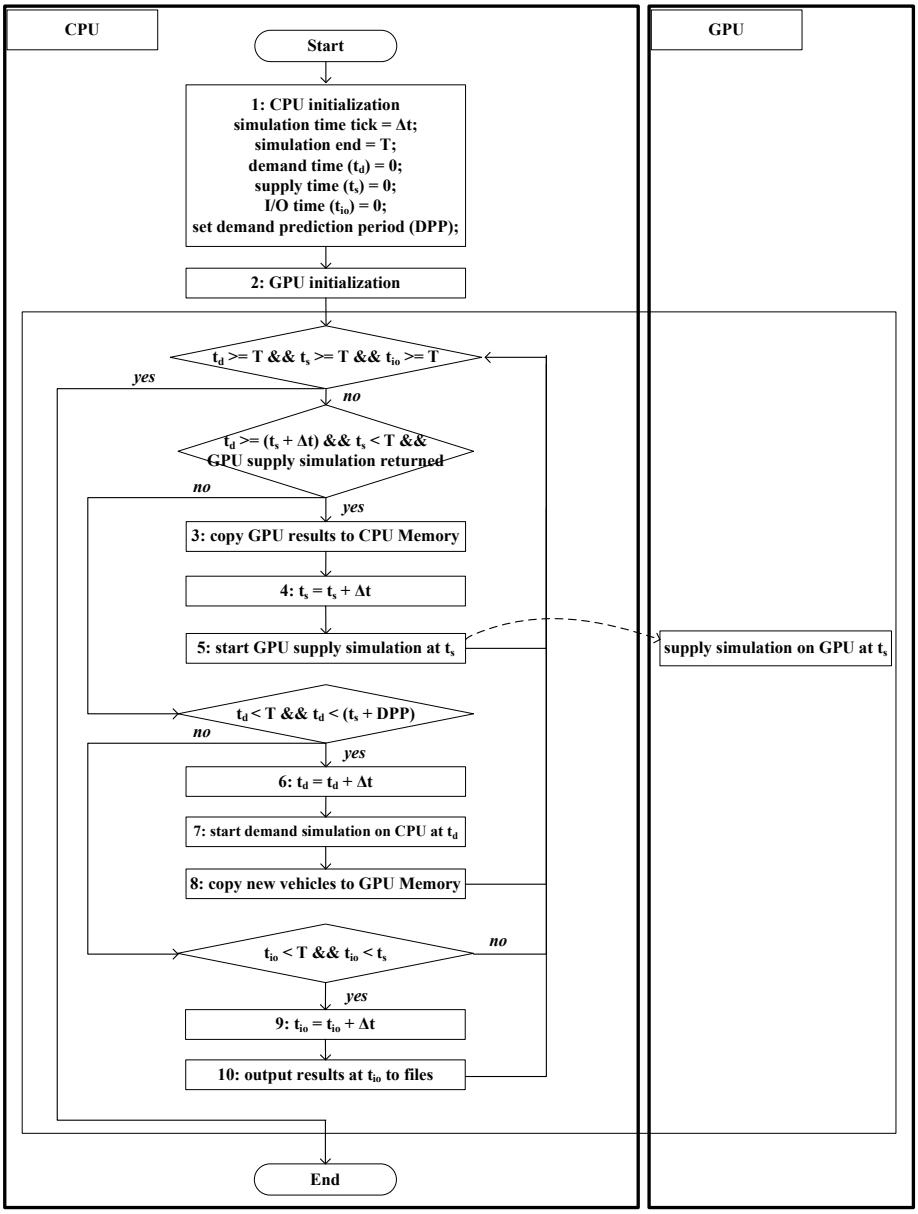


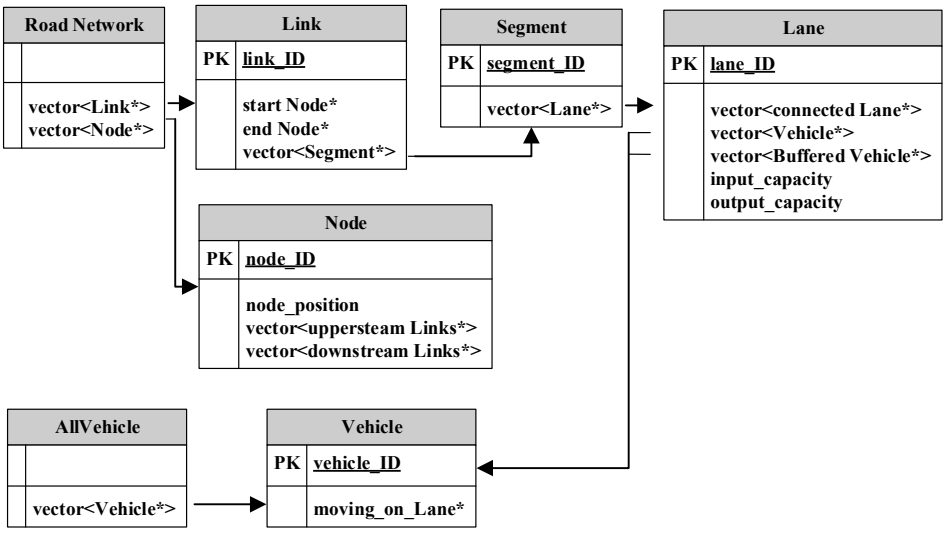
Figure 3. Mesoscopic Traffic Simulation Framework on CPU-GPU

involves a concept in traffic simulation: demand prediction period (DPP). Vehicles generated at time *t* requires the simulated results at *t* − DPP, for departure time choices and route choices. The minimum value of DPP is 1, which means vehicles have real-time instantaneous information about the global traffic status in last time step (e.g. 1 second). However, DPP tends to be larger in real-world traffic systems (e.g. 15 minutes).

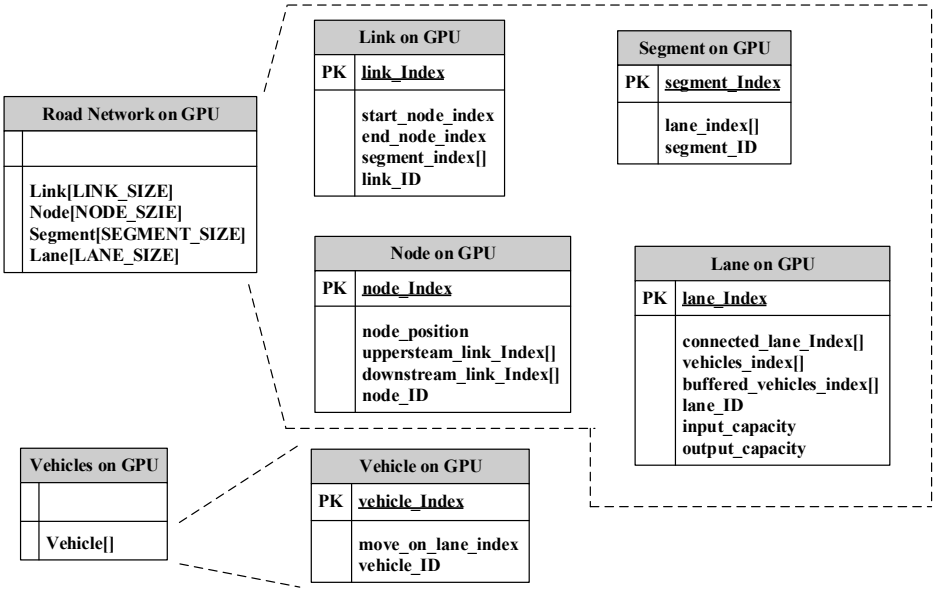
In the logic procedure in Figure 3, step 1 and 2 initialize the required data structures on the CPU and the GPU, including the road network, traffic scenario configurations and other parameters. After initialization, the CPU controls the simulation logic, in order to manage the simulation time and also to make full use of computational resources. Without breaking the three rules in time management, the following tasks can be executed in parallel:

- Task 1: The supply simulation at time t_s on GPU (step 3-5).
- Task 2: The demand simulation at time t_d on CPU (step 6-8).
- Task 3: Push simulation results at time t_{io} to files (step 9-10).

Within a loop of the logic procedure, the CPU firstly checks whether the GPU has finished the supply simulation at time t_s. If yes, the simulation results on the GPU (e.g. road-based speed and density) are copied to the CPU and the supply time t_s is advanced. Then, the supply simulation at next time step is started on the GPU. Note that the CPU will not wait for the GPU supply simulation to finish. If the supply simulation on the GPU is ongoing, the CPU checks whether the demand simulation can be started. If the simulation results required for demand simulation are available, the demand simulation will be started on the CPU. Otherwise, the CPU checks whether there are available simulation results that need to be written into files. The CPU will continue the loop until the three time t_s, t_d and t_{io} are all reaching the simulation end. The logic of the supply simulation and the demand simulation are explained in [1, 16].



(A) Road network and vehicle modeling on the CPU



(B) Road network and vehicle modeling on the GPU

Figure 4. Key differences in road network and vehicle modeling on the CPU and the GPU

3.2 Road Network and Vehicle Modeling on the GPU

Figure 4 (A) shows the road network and vehicle modeling on the CPU memory. A road network is composed of a list of links and a list of nodes. Each link consists of a number of segments and each node consists of a list of upstream links and downstream links. Each segment consists of multiple lanes and each lane contains a number of lane connections. Vehicles are moving on lanes or segments. Figure 4 (B) shows a similar road network and vehicle modeling on the GPU memory. Note that the purpose is to show the difference of network modeling between CPU memory and GPU memory. Figure 4 contains only a portion of network-related parameters.

There are two key differences between the road network modeling in the CPU memory and the GPU memory. **First**, on the CPU memory, the large number of road elements and vehicles are stored in random separated memory spaces and the objects are connecting with each other using *pointers* (or memory addresses). While on the GPU memory, these elements are kept in *arrays* in a continuous memory space and different elements are connecting each other using the *index* inside the *array*. The reasons of doing this on the GPU memory include making it easy to copy the entire road network from the CPU memory to the GPU memory and more importantly to allow efficient *coalesced memory access*, which means GPU threads in a warp tend to access continuous memory space. **Second**, on the CPU memory, dynamic memory allocation (e.g. std::vector) is widely used in the data structure of a road network and vehicles, because of its flexibility and efficiency. However, on the GPU memory, dynamic memory allocation (e.g. std::vector) has to be replaced by fixed memory allocation (e.g. array). It is a limitation of GPU programming, because it is not efficient to dynamically malloc and release GPU memory in kernel/device functions. It generates some memory issues. For example, as shown in Figure 4 (B), each lane has an array named

"vehicle_index", which contains the index of all vehicles located on the lane. The size of the array is the maximum number of vehicles the lane has space for. The memory space for the array is mandatory, even if there is no vehicle moving on the lane during the traffic scenario.

3.3 Kernel Functions on the GPU

There are four kernel functions in supply simulation on the GPU:

- Kernel function 1: *check_entrip_route_choice*
- Kernel function 2: *pre_vehicle_passing*
- Kernel function 3: *vehicle_passing*
- Kernel function 4: *copy_simulation_results_to_cpu*

The first kernel function identifies vehicles that require en-trip route choices, because of immerging traffic conditions (e.g. traffic congestions). The update unit of this kernel function is a vehicle, which means each individual vehicle is updated using a GPU thread. If a vehicle decides to change its route, it sets an attribute "entrip_route_choice" to be true. En-trip route changing behavior is simulated on the CPU and then the new routes are copied back to the GPU. As explained in Section 3.2, the attribute of all vehicles are kept in a continuous memory space, in order to minimize the communication cost between the CPU and the GPU. The first kernel function involves the CPU program to update the routes, thus, it is costly. There are three methods to reduce the cost. Firstly, if the en-trip route choice behavior is not required in the traffic scenario, this kernel function can be disabled. Second, in most traffic scenarios, there is no need to check en-trip route choice at each time step. Third, if a vehicle's route changing decision does not immediately change his status in the next time, the GPU does not need to wait for the end of this kernel function.

The second kernel function updates the status of each road (e.g. density, speed and t_p), before passing vehicles to the downstream roads. The update unit of this kernel function is a road (e.g. a lane), which means each individual road is simulated on a GPU thread. This kernel function firstly inserts vehicles, which passed to this road at the previous time, into the road. After that, it loads new generated vehicles into the road. Then, the kernel function calculates speed of the road based on a speed density relationship. After that, it calculates the *entry_time_to_pass*. The calculation algorithms are explained in [16].

The third kernel function scans vehicles on the road and passes some vehicles to the downstream roads. If a vehicle is moved from the current road to a downstream road, the corresponding output capacity, input capacity and empty space should be updated. The index of the vehicle should be removed from the current road and inserted to the target road since vehicles from links, which have the same end node, might conflict each other, if they are moving to the same downstream link. Each node, including its upper stream links, is considered as a basic process unit and is updated in a separate GPU thread. One example is shown in Figure 5. In this small road network, node 1 has two upstream links: link 1 and link 4. Thus, vehicles on link 1 and link 4 are processed on the same GPU thread, to remove the potential conflicts. On the other hand, since each link has only one upstream node, from where vehicles might pass through, there is also no conflicts when updating nodes in massive parallel. After passing vehicles to downstream roads, this kernel function updates the queue length, t_q and empty space of each road. The calculation of queue length is also explained in [16].

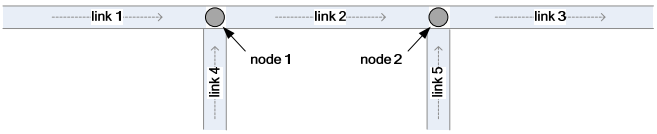


Figure 5. Each node and its upstream links are updated on the same GPU thread

The last kernel function copies the simulated results, which include speed, density, flow, queue length and empty space of each road, from the GPU memory to the CPU memory. As shown in Figure 4(B), these data are stored in a contiguous GPU memory space, in order to reduce the time cost of data transfer from the GPU memory to the CPU memory.

3.4 Incident Simulation

Incident simulation captures the impact of traffic incidents on the road capacity and traveler's response during a period, and predicts future traffic conditions, by simulating the interaction between the changed traffic demand and the reduced traffic supply. Incident simulation is critical for simulation-based traffic planning and simulation-based traffic management. In this framework, an incident is characterized by the ID of the affected road, incident start time and end time and a capacity reduction factor (e.g. 40%) on the road [4].

When simulating a traffic incident on CPU/GPU, the CPU reads the incident information from offline files (for offline traffic planning) or real-time data channel (for online traffic prediction). When the supply simulation time t_s reaches the incident start time, the CPU reduces the capacity reduction factor of the corresponding roads and then updates the road capacity in the GPU memory before starting the traffic simulation on the GPU at the next time step. When the supply simulation time t_s reaches the incident end time, the CPU recovers the road capacity and then updates the original capacity in the GPU memory.

During an incident period, vehicles moving on the road network might change their routes, based on their perceived traffic information. The drivers' intention to change routes is simulated in the first kernel function. If a vehicle decides to change its route, the new route is calculated on the CPU. Note that the current design involves additional data copy between the CPU and the GPU, which is time costly in current NVIDIA architectures [10]. However, the cost will be reduced in future architectures when the CPU and the GPU share the same memory space. Compared to en-trip route choice, pre-trip user behaviors are more efficient. The CPU is responsible to choose the updated departure time and the updated routes for drivers, taking into consideration the incident simulation results from the GPU. There is no additional data copy between the CPU and the GPU.

4. CORRECTNESS

The purpose of running mesoscopic supply simulation on the GPU is to boost its massive computational power. However, it is critical to guarantee that the supply simulation on GPU gives the same results as the supply simulation on CPU, or at least the difference is acceptable.

4.1 Problem Definition

A challenging problem is that mesoscopic traffic simulations in a road network cannot be naturally spatially divided into multiple independent traffic simulations in a large number of sub-networks, because of *upstream downstream dependence* when vehicles cross the boundaries of sub-networks. An example is shown in Figure 6. When a vehicle crosses from link 1 to link 2, there are three conditions to check:

1. Whether there is available output capacity at link 1.
2. Whether there is available input capacity at link 2.
3. Whether there is enough empty space at link 2.

As explained in the third kernel function in Section 3.3, each individual node and its upstream links are simulated on a GPU thread. Before starting a supply traffic simulation on the GPU, the pre-configured input capacity and output capacity of links are available. Besides, the input capacity of each link is updated by only 1 node (or 1 GPU thread). It means the capacity of links does not cause problems when running mesoscopic traffic simulations on the GPU. However, the empty spaces at downstream links depend on the traffic movement of downstream links. The requirement to know the empty space of the downstream links when simulating traffic movement in the upstream links is defined as *upstream downstream dependence*. When running mesoscopic traffic simulations in a sequential way, *upstream downstream dependence* is complied by ordering the links from downstream to upstream. For this example in Figure 6, it means that in the mesoscopic supply simulation, traffic movement on link 2 is simulated before traffic movement on link 1. If there is a road circle in the road network, the road circle can be break at a random node. However, when a mesoscopic supply simulation is spatially divided into multiple traffic simulations in a number of sub-networks, it is a challenge to comply with all *upstream downstream dependencies.*

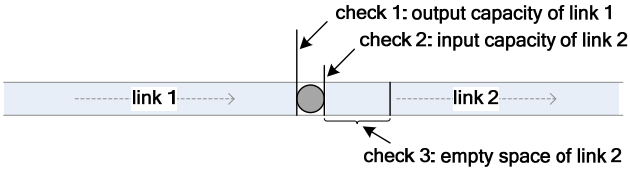


Figure 6. An example upstream downstream dependence in mesoscopic traffic simulations

When perfect knowledge of the empty space in a downstream road is not available, the upstream link has to move vehicles based on its best estimate. This could lead to two types of unrealistic vehicle movements [2]:

Pessimistic biased movement: If vehicles' movement in an upstream links is based on overly conservative estimate of the downstream empty space, when empty space of downstream link at time t is estimated to be the empty space of that link at the end of t-1, vehicles might move slower than the sequential simulation. For example, if a downstream link is blocked at time t-1, an upstream link may assume that the downstream link will be still blocked at time t, which might be wrong. It will force upstream vehicles to stay on the upstream link.

Optimistic biased movement: An upstream link could over-estimate the empty space available at a downstream link. For instance, it may optimistically assume a downstream link always has enough empty space for vehicles. In this case, the simulator may fail to capture exactly the same queuing and spill-backs from the sequential traffic simulation.

Moving vehicles on upstream links based on estimated empty spaces on downstream links in parallel might generate simulation errors (compared to a sequential traffic simulation), when simulating congested traffic scenarios. In free-flow traffic scenarios, the empty spaces of most links are large and the speeds on roads are high. Thus, empty spaces are less likely to be a limitation for vehicles on upstream links to cross. In this case, optimistic vehicle movement can get almost the same results with the sequential simulation. However, in congested traffic scenarios, the empty spaces of links can be small (or even zero). It becomes critical to have an accurate estimation of empty spaces in downstream links, in order to decide whether or not to pass a vehicle.

4.2 Boundary Processing Method for Massive Parallelism on the GPU

The concept of boundary processing has been previously used in spatial parallel traffic simulations [2]. The boundary area means a portion of a road network which connects the traffic flow from different road partitions. In most cases, traffic movement in a boundary area requires a different procedure compared with traffic movement on a normal road. In this paper, when running traffic supply simulation on the GPU and each node of a road network is simulated on a separate GPU thread, the boundary area is in fact the global road network.

The purpose of the boundary processing method is to support an accurate estimation of empty spaces in downstream links, before moving vehicles to cross links. Thus, the simulation results from a massive parallel traffic simulation on the GPU are similar to the sequential simulation. As shown in formula 1, the update of an empty space on a road at a time step t (ES(t)) depends on three variables: the empty space of the road at the previous time step t-1, the speed of the road at time t ($v(t)$) and the queue length of the road at t ($q(t)$).

$$ES(t) = \min\{ES(t\text{-}1) + v(t)\ ,\ RL - q(t)\} \quad (1)$$

Where, RL is the length of the road, $ES(t\text{-}1) + v(t)$ reflects the traffic movement on the road and $RL - q(t)$ reflects the feedback of the queue on the road.

The proposed boundary processing method consists of two steps. First, an additional synchronization (or a barrier) is inserted between speed calculation and vehicle movement. It does not affect the simulation results. Since empty spaces are calculated in the phase of vehicle movement, the speed of all roads turns to be available before empty space calculation. As explained in Section 3.3, the speed calculation happens in the second kernel function and the empty space calculation happens in the third kernel function. These two operations are naturally separated. Thus, the synchronization does not bring additional cost to the framework. Besides, when this paper is written, the time cost to switch between different kernel functions on the GPU is trivial. Second, the queue length of each road in the current time t is predicted before empty space calculation using formula 2.

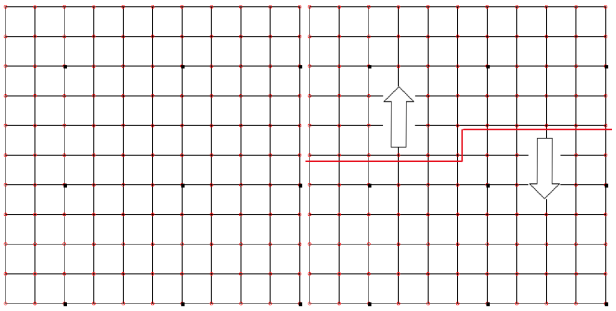
$$q(t) = \max\{q(t\text{-}1) + a * (q(t\text{-}1) - q(t\text{-}2)),\ RL\} \quad (2)$$

Where RL is the length of the road, (q(t-1) - q(t-2)) reflects the short-term trend of the queue length and a is a parameter, which

indicates how much the predicted queue length depends on the short-term trend.

4.3 Evaluation

This section evaluates the efficiency of the boundary processing method. As shown in Figure 7(A), a grid road network is used in this section. There are 121 nodes and 220 unidirectional links in the road network. The length of each link is 1000 meters. Each node has an ID, from 0 to 120. For example, the nodes in the first (top) row have IDs from 0 to 10. Vehicles are loaded into the road network from nodes in the top and the left, which are moving to the bottom and the right. When a vehicle is loaded into the road network, the vehicle randomly chooses a route from pre-calculated candidate routes. Vehicles on the same road are moving using the same speed, which is calculated using a linear speed-density relationship [16]. 50,000 vehicles are loaded into the road network within an hour. As the traffic simulation is on-going, the road densities in the bottom right corner turns to be higher than the road densities in the top left corner. The basic simulation time step is 1 second, which means the system status is updated every 1 second and there are 3600 simulation ticks in this traffic scenario. Speed, density, flow, queue length and empty space of each link at each time are outputted to files as the simulation results.



(A) (B)

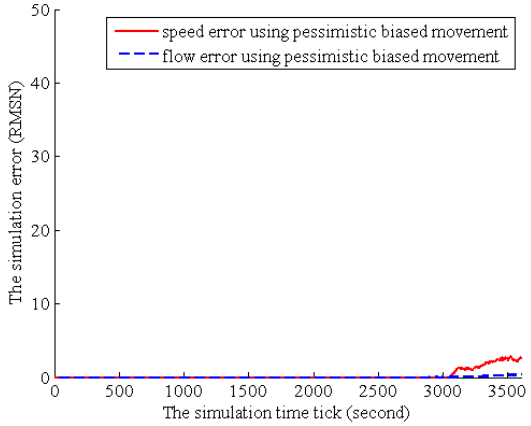
Figure 7. (A) a grid road network with 121 nodes and 220 unidirectional links; (B) the grid road network is divided into two partitions, running on 2 threads.

In a sequential traffic simulation, nodes (and the upstream links) are updated in order (from downstream to upper stream). For example, nodes in the bottom-right corner are updated before nodes in the top-left corner. All *upstream downstream dependencies* are followed in a sequential traffic simulation. Comparatively, the grid network will be divided into a number of sub-networks in a parallel traffic simulation. As shown in Figure 7 (B), the grid road network is divided into two partitions. Nodes in each individual partition are updated in order, to follow the *upstream downstream dependencies* within the partition. But there is no guarantee of the *upstream downstream dependencies* between these two partitions. For the vehicle movement in upstream links nearby the boundary, this section evaluates two methods: the pessimistic biased movement and the proposed boundary processing method. Since the traffic scenario is congested, the optimistic biased movement is not suitable. The simulation results (road-based speeds and densities) are compared with the sequential traffic simulation to measure the correctness at each simulation time tick. The performance indicator is the Normalized Root Mean Square error (RMSN), which measures the proportional difference between two vectors [4].

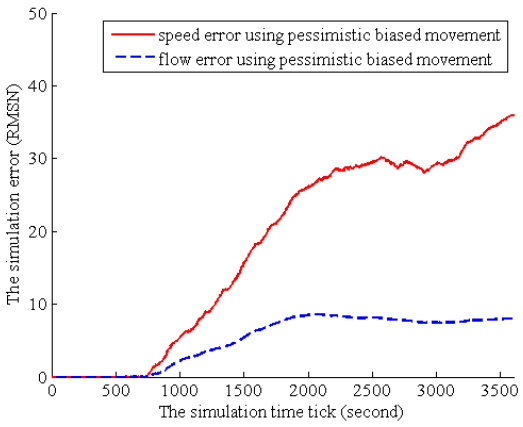
$$RMSN = \frac{\sqrt{N * \sum_{i=1}^{N} (s_i - p_i)^2}}{\sum_{i=1}^{N} s_i} \quad (3)$$

Where, N is the length of vectors, S_i is a simulation result of the sequential traffic simulation and p_i is a simulation result of the parallel traffic simulation.

The results (simulation correctness) are shown in Figure 8. Figure 8(A) shows the speed and density simulation error using pessimistic biased movement when the simulation is divided into 2 partitions. As shown in Figure 7(B), the boundary area is in the middle of the road network. It is a surprise to see the simulation results of the parallel traffic simulation using pessimistic biased movement are exactly the same with the sequential traffic simulation before the 2800 tick, even though it is using inaccurate empty spaces. We found that in this traffic scenario, congestion starts in the bottom area and gradually spills back to the middle area of the road network. It indicates the pessimistic biased movement might give an acceptable correctness when parallel traffic simulations are run on a small number of partitions and there are no traffic congestions around the boundary area. When traffic congestion spills back to the boundary area, a maximum 2.9% speed error and a maximum 0.4% flow error are observed in this case study. Figure 8(B) shows the speed and density simulation error using pessimistic biased movement when the simulation is divided into 121 partitions, which is the case to run the mesoscopic supply simulation on the GPU. The simulation error using pessimistic biased movement happens when congestion starts in the bottom area and grows fast when the congestion spills back to the middle of the road network. Finally, a maximum 36.0% speed error and a maximum 8.6% flow error are observed in this case study. It means that the pessimistic biased movement gives an unacceptable correctness when running congested traffic scenarios on the GPU in a massive parallel way. On the other side, for these two cases, the proposed boundary processing method gets exactly the same simulation results with the sequential traffic simulation, or the speed error and the flow error during the whole simulation period are 0%.



(A)



(B)

Figure 8. (A) The speed and density simulation error when divided into 2 partitions using pessimistic biased movement. (B) The speed and density simulation error when divided into 121 partitions using pessimistic biased movement. (Note: there is no speed and density simulation error in A and B when using the proposed boundary processing method)

Even though the experiment results prove the efficiency of the proposed boundary processing method, there are two limitations. First, the quantitative results from this experiment (e.g. 2.9% and 36%) are not significant and it is difficult to directly generalize the quantitative results to other mesoscopic traffic simulations on other road networks. The road network topology and the behavior models (e.g. the speed-density relationship) in mesoscopic traffic simulations might change experiment results. Second, there is no theoretical guarantee for the proposed boundary processing method to always get the same results with the sequential traffic simulation, because the predicted queue length might be wrong in the boundary processing method.

5. Double-Buffer Data Channel on the GPU

The process of copying simulation results from the GPU memory to the CPU memory at each time step (step 3 in Figure 2) is time costly in CPU/GPU. As shown in Figure 9, a double-buffer data channel is designed on the GPU to minimize the data communication cost. There are two optimizations in the double-buffer data channel. Firstly, the frequency to copy simulation results from the GPU memory to the CPU memory is reduced. For example, in this case, each buffer can keep simulation results of up to 8 time steps (known as "*buffer size*"). When the buffer is full, the supply simulation on the GPU starts to write simulation results to the other buffer space. At the same time, the simulation results in the buffer are copied to the CPU memory in one time step. Secondly, the data transfer from the GPU to the CPU is done asynchronously. It means the supply simulation on the GPU does not wait for the data transfer to finish. However, the double-buffer data channel brings two additional rules in time management.

- Time management rule 4: *buffer size* <= DPP
- Time management rule 5: $t_s < t_{io} + 2 * buffer\ size$

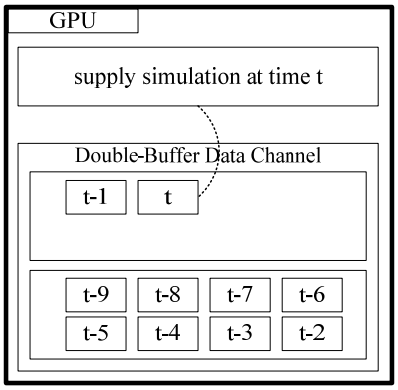


Figure 9. An example double-buffer data channel on GPU (the buffer size is 8)

If the buffer size is larger than the demand prediction period (DPP), there will be a deadlock in time management. The buffer is waiting for additional supply simulation results before transfer its data to the CPU. At the same time, the demand simulation on the CPU is waiting for the simulation results from the GPU to generate future vehicles and the supply simulation on the GPU is blocked because of the lack of new vehicles. The mutual waiting between the three components is a deadlock. The suggested *buffer size* should be much smaller than DPP. In this paper, DPP is 15 minutes and *buffer size* is 1 minute (or 60). Besides, rule 5 says the supply simulation on the GPU cannot write simulation results to a buffer, if the data in the buffer has not been transferred to the CPU. It means a slow data transfer or I/O will finally force the supply simulation on the GPU to stop and wait.

6. Experiments

In this section, the proposed mesoscopic traffic simulation framework on CPU/GPU is implemented and evaluated to simulate a large-scale traffic scenario. This section focuses on the performance comparison of the supply simulation on the CPU and the GPU.

6.1 Testbed

An artificial large grid road network, which is similar to the road network in Figure 7(A), is used as the testbed, with 10201 nodes and 20200 unidirectional links. Each node has an index from 0 to 10200, indicating the store location on the GPU memory. Each link also has an index from 0 to 20199. 100,000 vehicles are loaded into the road network during 1000 simulation ticks (each tick is 1 second). Same with the experiment configuration in Section 4.3, vehicles are loaded into the road network from nodes in the top and the left, which are moving to the bottom and the right. Each vehicle randomly picks a route from the pre-calculated candidate routes before starting a trip. En-trip route choice is not included in this traffic scenario.

The traffic scenario is simulated on two types of platforms: the CPU and CPU/GPU. Only the total time cost of the supply simulation during the 1000 simulation ticks is measured in this experiment. The CPU platform includes an Intel E5-2620, 32 GB main memory and a 500GB SATA 7.2K RPM. The GPU platform is a GeForce GT 650M, which has 384 CUDA cores and 2 GB global memory. The supply simulations on the CPU and the GPU follow the same logic. The source codes are both implemented using C++ on Ubuntu 12.04 and compiled using

g++_4.6.3 and CUDA 5.5. The release version executable file is used to measure the time cost.

6.2 Results and Analysis

The experiment results are shown in Table I. The time cost column shows the average time cost of 5 different measurements of each configuration. The speedup is measured by comparing the time cost of supply simulation on a GPU to the time cost of supply simulation on a CPU core. In the first configuration, executing the supply simulation on a CPU core takes 4720.88 ms to finish the traffic scenario. In the second configuration, directly executing the proposed supply simulation on a GPU takes 704.72 ms to finish the same traffic scenario, which means the speedup is 6.7. The performance is sensitive to the configuration of the kernel functions. When the number of threads in a block is 192, the maximum performance is achieved. In the third configuration, the double-buffer data channel on the GPU is enabled to allow asynchronous data transfer between the CPU and the GPU. We found that the double-buffer data channel is efficient and the data transfer cost is almost hidden by the supply simulation on the GPU. The total time cost is reduced by 35% and the speedup is significantly improved from 6.7 to 10.3. Besides, we found that the registers are not efficiently used in the third configuration. In the fourth configuration, internal variables, which will not be used by other kernel functions and not transferred to the CPU, are moved from the global memory to the registers. The speedup is slightly improved from 10.3 to 10.7. Finally, there are parameters which are never changed during the traffic simulation, such as the simulation end time and settings in the speed-density relationship. In the fifth configuration, constant parameters are moved from the global memory to the constant memory for efficient memory access. The speedup is slightly improved from 10.7 to 11.2.

Table I: the time cost of running a traffic supply simulation on the CPU and the GPU

Case ID	Configuration Description	Time Cost (ms)	Speedup
1	Supply simulation on a CPU core	4720.88	1.0
2	Supply simulation on a GPU	704.72	6.7
3	Case 2 + Double-buffer data channel optimization on the GPU	457.29	10.3
4	Case 3 + Push internal variables from the global memory to the registers	439.29	10.7
5	Case 4 + Push constant parameters from the global memory to the constant memory	423.37	11.2

Table II shows major profiling measurements of two major kernel functions in the framework: *pre_vehicle_passing* and *vehicle_passing*. First, the processing unit in the first and the second kernel function are a road and a node. In this experiment, the two kernel functions launched 20200 GPU threads and 10201 GPU threads. The occupancies of these two kernel functions are high, which indicates the GPU cores are fully utilized. Though internal variables are moved from the global memory to the registers, registers are still not a bottleneck. Second, as explained in Section 3.2, the network data and vehicle data are stored into a contiguous memory space. Thus, threads in a warp tend to access a contiguous memory space, which is also known as *coalesced memory access*. As shown in the table, the number of memory transaction per request (both load and store) for these two kernel functions are small, which indicates the memory load and store is efficient. Third, the branch taken ratio (within threads in the same warp) for the first kernel function is 72%. Threads in the first kernel function do not take exactly the same branch, because different roads have different number of new vehicles and different number of passed vehicles. The branch taken ratio for the second kernel function is much lower (41.7%). It is expected, because in the second kernel function there are many for-loops in the logic of finding and passing vehicles to downstream roads. The number of upstream links and the number of vehicles on nodes are different. However, there is no much branch divergence in these two kernel functions. The instruction serialization ratio of these two kernel functions are 15.5% and 18.6%. It means that the low branch taken ratio is not a big problem in this framework. Fourth, the numbers of instruction per clock (IPC) for the two kernel functions are 0.9 and 1.0, which are far below the hardware's peak value (4.0). As shown below, the major reason for low issue efficiency is the execution dependency. Based on our knowledge of the framework, the major problem is that most data (e.g. the road network and vehicles) is stored in the global memory, which causes high memory latency. The high memory latency cannot be completely hidden by the large number of threads. Finally, the achieved GLOPS for the two kernel functions are also lower than the hardware's peak, which is also related with high global memory latency.

Table II: The profiling of major kernel functions in mesoscopic traffic simulations

	Parameters	pre_vehicle_passing	vehicle_passing
1	Launched GPU threads	20200	10201
2	GPU occupancy	81%	90%
3	Registers (used / available)	3072 / 65536	1920 / 65536
4	Transaction per request (load/store)	1.73/1.49	1.67/1.83
5	Branch taken ratio (%)	72.1%	41.7%
6	Instruction serialization	15.5%	18.6%
7	Instruction per clock (IPC) (measurement/ maximum)	0.9 / 4.0	1.0 / 4.0
8	Warp issue efficiency (no eligible %)	49.7%	36.1%
9	Issue Stall Reasons (execution dependency)	92.3%	88.4%
10	CUDA achieved GFLOPS	14.8	6.0

This paragraph discusses additional thoughts about running mesoscopic traffic simulations on CPU/GPU. First, it is beneficial to run the demand simulation on the CPU, the supply simulation on the GPU and the data communication between the CPU and the GPU in an asynchronous way. In this experiment, the supply simulation on GPU is the bottleneck and the time costs of the other two tasks are almost hidden. Second, this paper demonstrates a supply simulation framework (ETSF) on the GPU. Running the ETSF framework on the GPU gets a speedup of 11.2, compared with running the same logic on the CPU. However, when generalizing the conclusion to other supply simulation frameworks, it should be noted that ETSF naturally guarantees a good load balance on each road. In ETSF, the workload of a road is not sensitive to the number of vehicles on the road and the length of the road. This feature might do not exist in other supply simulation frameworks, in which case, load balance should be considered. Third, the memory access latency is a bottleneck in the proposed mesoscopic traffic simulation framework. In current mesoscopic supply simulation

frameworks [1, 4, 5, 16], the update logic of a road majorly depends on its own road status (e.g. road density and queue status) and requires a small number of parameters from its downstream roads. There is few shared data access among nearby roads and nodes, which limits the usage of the efficient shared memory in the GPU. It is the major reason of high memory latency.

7. CONCLUSIONS AND FUTURE WORK

Mesoscopic traffic simulation is hungry for computational resources to support aggressive large-scale simulation-based traffic planning and simulation-based traffic management. The GPU has recently been a success, because of its massive performance compared to the CPU. Thus, a critical question is "whether the GPU can be a potential high-performance platform for future mesoscopic traffic simulations?" In this paper, we proposed a comprehensive mesoscopic traffic simulation framework on CPU/GPU. Then, we designed an innovative boundary processing method to guarantee the correctness of running massive parallel mesoscopic traffic simulations on the GPU. The proposed mesoscopic traffic simulation framework is evaluated to simulate a large-scale traffic scenario and gets a speedup of 11.2, compared with running the same logic on the CPU. Based on our view, the proposed mesoscopic traffic simulation framework on CPU/GPU provides an innovative and promising solution for high-performance mesoscopic traffic simulations.

Further research on the topic of mesoscopic traffic simulation on CPU/GPU includes two directions. First, the proposed mesoscopic traffic simulation framework needs to be evaluated to simulate a real-world large-scale traffic scenario. Second, the proposed mesoscopic traffic simulation framework needs to be improved to make better use of the shared memory in the GPU.

8. ACKNOWLEDGMENTS

The authors would like to thank Kakali Basak, Stephen Robinson, Lu Yang, Francisco Pereira and Harish Loganathan for their comments to this paper.

9. REFERENCES

[1] Barcelo J. (editor), "Fundamentals of Traffic Simulation", International Series in Operations Research & Management Science, 2010, Springer, New York.

[2] Yang W., "Scalability of Dynamic Traffic Assignment", Ph.D. thesis, Massachusetts Institute of Technology, 2009.

[3] Ben-Akiva, M., Gao, S., Wei, Z. and Yang, W., "A dynamic traffic assignment model for highly congested urban networks", Transportation Research Part C, 2012, 24: 62–82.

[4] Ben-Akiva, M., Bierlaire, M., Burton, D., Koutsopoulos, H. N., and Mishalani, R., "Network state estimation and prediction for real-time traffic management", Networks and Spatial Economics, 2001, 1(3/4):293-318.

[5] Mahmassani H.S., Hu T., and Jaykrishnan R., "Dynamic traffic assignment and simulation for advanced network informatics (DYNASMART)", Proceedings of the 2nd International Capri Seminar on Urban Traffic Networks, Capri, Italy, 1992.

[6] Ziliaskopoulos A. K., Waller S. T., Li Y., and Byram, M., "Large-scale dynamic traffic assignment: Implementation issues and computational analysis", Journal of Transportation Engineering, 2004, 130(5): 585-593.

[7] Gordon D. B. C. and Gordon I. D. D., "Paramics - Parallel Microscopic Simulation of Road Traffic". The Journal of Supercomputing, 1996, pp. 25-53.

[8] Çetin, N., "Large-scale parallel graph-based simulations", Ph.D. thesis, ETH Zurich, Switzerland, 2005.

[9] Aydt H., Yadong X., Michael L., and Alois K., "A Multi-threaded Execution Model for the Agent-Based SEMSim Traffic Simulation", Proceedings of AsiaSim 2013.

[10] Nvidia, "CUDA C Programming Guide", 2013, available at http://www.nvidia.com/CUDA.

[11] Michalakes J. and Vachharajani M., "GPU acceleration of numerical weather prediction", IPDPS 2008: IEEE Int'l Symp. Parallel and Distributed Processing, 2008.

[12] Buck I., "GPU computing with NVIDIA CUDA". In SIGGRAPH '07, New York, NY, USA, 2007.

[13] Fan, Z., Qiu, F., Kaufman, A., and Yoakum-Stover, S., "GPU cluster for high performance computing". In Proceedings of SC'04, 2004.

[14] Perumalla, K. S., Brandon G. A., Srikanth B. Y., and Sudip K. S., "GPU-based real-time execution of vehicular mobility models in large-scale road network scenarios", International Workshop on Principles of Advanced and Distributed Simulation, 2009.

[15] Strippgen, D. and Nagel, K., "Multi-agent traffic simulation with CUDA", Proceedings of International Conference on High Performance Computing & Simulation, Leipzig, 2009.

[16] Yan X., Xiao S., Zhiyong W. and Gary T., "An Entry Time based Supply Framework (ETSF) for Mesoscopic Traffic Simulations", submitted to Simulation Modelling Practice and Theory, 2014.

[17] Denis G., Jose-Juan T., Samuel A. and Roshan M. D. S., "Graphics processing unit based direct simulation Monte Carlo", Simulation: Transactions of the Society for Modeling and Simulation International, 2012, 88(6): 680-693.

[18] Brodtkorb A. R., Trond R. H. and Martin L. S., "Graphics processing unit (GPU) programming strategies and trends in GPU computing", Journal of Parallel and Distributed Computing, 2013, 73: 4-13.

[19] Perumalla, K. S., "Discrete Event Execution Alternatives on Gen-eral Purpose Graphical Processing Units (GPGPUs)", International Workshop on Principles of Advanced and Distributed Simulation, 2006.

[20] Passerat-Palmbach, J.; Mazel, C.; Hill, D. R C, "Pseudo-Random Number Generation on GP-GPU," International Workshop on Principles of Advanced and Distributed Simulation, 2011.

[21] Xiaosong L., Wentong C. and Stephen J. T., "GPU Accelerated Three-stage Execution Model for Event-parallel Simulation", International Workshop on Principles of Advanced and Distributed Simulation, 2013.

[22] Park H. and Fishwick P. A., "An analysis of queuing network simulation using GPU-based hardware acceleration", ACM Transactions on Modeling and Computer Simulation. 2011, 21(3).