High-Performance Mesoscopic Traffic Simulation with GPU for Large Scale Networks

Vinh An Vu
School of Computing
National University Singapore
Singapore 117417
Email: vuvinhan01@u.nus.edu

Gary Tan
School of Computing
National University Singapore
Singapore 117417
Email: dcstansh@nus.edu.sg

Abstract—**Mesoscopic Traffic Simulation is an important tool in traffic analysis and traffic management support. The balance between traffic modeling details and performance has made Mesoscopic Traffic Simulation one of the key solutions for traffic controllers and policy makers. Mesoscopic traffic simulators offer acceptable speed in simulating normal traffic. However, when traffic prediction and optimization for large scale networks come into context, the performance of mesoscopic traffic simulators is unsatisfactory in optimizing a massive number of control parameters for a much longer prediction horizon. This issue again emphasizes the need to further speed up mesoscopic traffic simulation. This paper proposes a comprehensive framework to massively speed up mesoscopic traffic simulation using GPU without compromising its correctness and realistic modeling property. It also gives an in-depth analysis into the trade-off between simulation correctness and performance speedup. By combining the power of GPU with optimal design and data structures, mesoscopic traffic simulation is able to speed up to more than 6 times compared to original CPU implementation.**

I. Introduction

The increasing power of computer technologies and the advent of Intelligent Transport Systems have turned traffic simulation to become one of the most used approaches for traffic analysis and evaluation[1]. Every traffic simulator models the supply side and the demand side. The supply side models the road network, traffic flow movement, traffic control, traffic management policy, incident, weather condition, etc. The demand side includes the modeling of drivers' origin-destination matrix and behavior. In traffic simulation, the level of modeling details depends on the modeler's purposes. Traffic flow can be modeled macroscopically, microscopically or mesoscopically. In macroscopic traffic simulation, traffic flows are modeled from an aggregated point of view and are characterized by aggregated variables such as speed, density and volume. In miscroscopic traffic simulation, the motion of each individual vehicle is modeled in detail with description of acceleration, deceleration, car following and lane change. In between the two former schemes is mesoscopic traffic simulation which combines microscopic aspects (modeling individual drivers) and macroscopic aspects (concerning only aggregated traffic properties).

Mesoscopic traffic simulation is computationally more efficient than microscopic counterpart. Many mesoscopic traffic simulators offer faster-than-real-time performance in simulating city-scale networks [2], [3]. In recent years, traffic simulation has come up with many advanced features such as prediction and traffic management strategy optimization. To make these new features useful in real-time traffic management support, faster-than-real-time is not enough. It is crucial that even thousands or more what-if simulations must be able to finish in a short time window to optimize a large number of control parameters. Super high-performance traffic simulation system is in urgent demand.

978-1-5386-4028-9/17/$31.00 ©2017 IEEE

In literature, parallel and distributed computing have mainly been used to speed up mesocoscopic traffic simulation, and traffic simulation in general. Existing studies have either employed multi-core CPU, clusters or GPU in their methodology. In using multi-core CPU and clusters, the general method is usually to partition the network into small sections, each is handled by one processing unit [4]. Besides the challenges in network partitioning, load balancing, synchronization and communication [5], the high hardware cost also hinders the attractiveness of this method.

On the other hand, GPU has recently moved out of its exclusive computer graphics zone to explore and accelerate many general-purpose computing problems ranging from artificial intelligence to cars, drones and robots. With more accessible programming interfaces and industry-standard languages such as C, usage of GPU in general purpose computing has become so ubiquitous that the term *gpgpu*, which stands for *General-Purpose computation on Graphics Processing Units*, has been coined [6]. But surprisingly, very few studies attempted to use GPU to speed up traffic simulation. Looking at the existing literature, the preliminary idea of using GPU to speed up field-based vehicular mobility model was initiated by Perumalla [7] in 2008. However, detailed formulation and result were not reported in that paper.

The first complete work in GPU-based traffic simulation was in 2009 with MATSIM [8]. By splitting vehicle movement into two kernels *moveNode()* and *moveLink()* to avoid synchronization and communication cost, the speed-up factor of up to over 60 times was achieved. This work is pioneering and innovative but it has two weaknesses. First, it ignored the handling of empty space in each link which relates to upstream-downstream dependency, therefore, compromising the simulation correctness. Second, the GPU implementation was compared against the Java implementation in CPU with significantly different simulation algorithms. This casts doubts on the impressive claimed speed-up factor.

Moving to 2011, Shen et. al. [9] made use of GPU in agent-based traffic simulation but the usage was limited to only the vehicle generation process.

In 2014, Xu et. al. [10] proposed a framework to run mesoscopic traffic simulation in CPU/GPU together with a boundary processing method to handle the upstream-downstream data dependency. The proposed framework achieved up to 11 times speed-up and guaranteed the correctness of simulation. However, the experiments were only conducted for a synthetic lattice network and the proposed boundary processing method was not guaranteed to work on all general networks. To fill in these gaps, this work was extended in [11] with the experiment in Singapore Expressway Network but the simulation correctness was still not investigated.

Another related recent work was by Sano et. al. [12] with focus on using GPU to speed up the demand side of traffic simulation (including path generation and agent negotiation) without much consideration for the supply side.

The review of existing work reveals two main gaps in GPU-based traffic simulation: (1) a systematic approach to fit mesoscopic traffic simulation in GPU with preserved simulation correctness is missing and (2) the trade-off between speed and simulation correctness was not analyzed in depth. We believe the deployment of GPU in traffic simulation deserves more attention and treatment. This paper proposes a comprehensive framework to massively accelerate mesoscopic traffic simulation using GPU with a thorough analysis of simulation correctness and speed-up factor. The contributions of this paper could be summarized in three points:

1) Two algorithms to fit mesoscopic traffic simulation in GPU architecture with optimal data structure, high level of data parallelism and different levels of simulation correctness.
2) An in-depth analysis of the trade-off between speed and accuracy.
3) Evaluation of the proposed algorithms in a large scale network of Singapore Expressways.

The remainder of this paper is structured as follows. Section II discusses the background information which serves as preliminaries for our work. Section III explains the core methodology and design of the framework. Section IV introduces in detail the implementation together with the data structures to optimize GPU performance. Section V showcases the experiments conducted in a real-life network and finally, Section VI concludes the paper with key ideas and proposal for future work.

II. Background

In this section, the preliminaries which laid foundation for our work will be discussed.

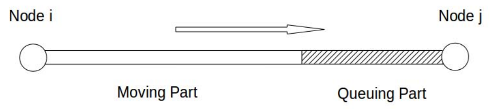

Fig. 1. Link Model

A. Mesoscopic Traffic Simulation

As introduced in Section 1, mesoscopic traffic simulation is one of the three modeling levels in traffic simulation. Basically, there are two main approaches to mesoscopic traffic simulation: one in which vehicles are grouped into platoons that move along the link, as in [13], and the other one in which dynamics of individual vehicles are simplified, as in [14], [2]. To handle time, mesoscopic traffic simulation either uses time-step or event-based model. Time-step model is computationally more expensive than event-based model. However, it is challenging to build an efficient event-based model especially for complex systems. In this paper, we deal with the time-step approach with individual vehicle modeling since it can better support applications for real-time traffic management [15].

A simulation network consists of nodes and links. Vehicles travel from one node (origin) to another node (destination) by passing through links. On the supply side of mesoscopic traffic simulation, links are modeled using queuing theory as in Figure 1. Each link is split into a queuing part and a moving part. Vehicles in the queuing part are delayed due to capacity constraints at the link downstream. On the other hand, nodes are modeled as the transfer of queuing vehicles from upstream flows to downstream ones ruled by the vehicle discharging process. To simulate traffic flow in the moving part, link density is computed based on number of vehicles on link and link physical length and speed is derived from density using *speed-density relationship* [1]. This speed is then used to calculate the time needed for vehicles in the moving part to reach the end of the link and be transferred to the next link.

The demand side of mesoscopic model includes vehicle loading model and route choice model in which the drivers select the route to travel in order to maximize some predefined objectives. Since the modeling of individual vehicles is simplified, mesoscopic simulation does not describe the interaction between vehicles such as lane changing and car following.

B. Graphics Processing Unit (GPU)

GPU is a specially designed electronic circuit to speed up computer graphic operations. It is commonly used as a parallel co-processor under the control of the host CPU in a heterogeneous system [16]. Even though originally built for computer graphics applications, GPU has emerged out of its comfort zone and become one important tool in massively parallel general-purpose programming over the last decade. Recently, CPU has hit its performance ceiling due to excessive power dissipation at GHz clock rates and diminishing returns in instruction-level parallelism while GPU has grown tremendously in performance due to massive parallelism. Hence, a

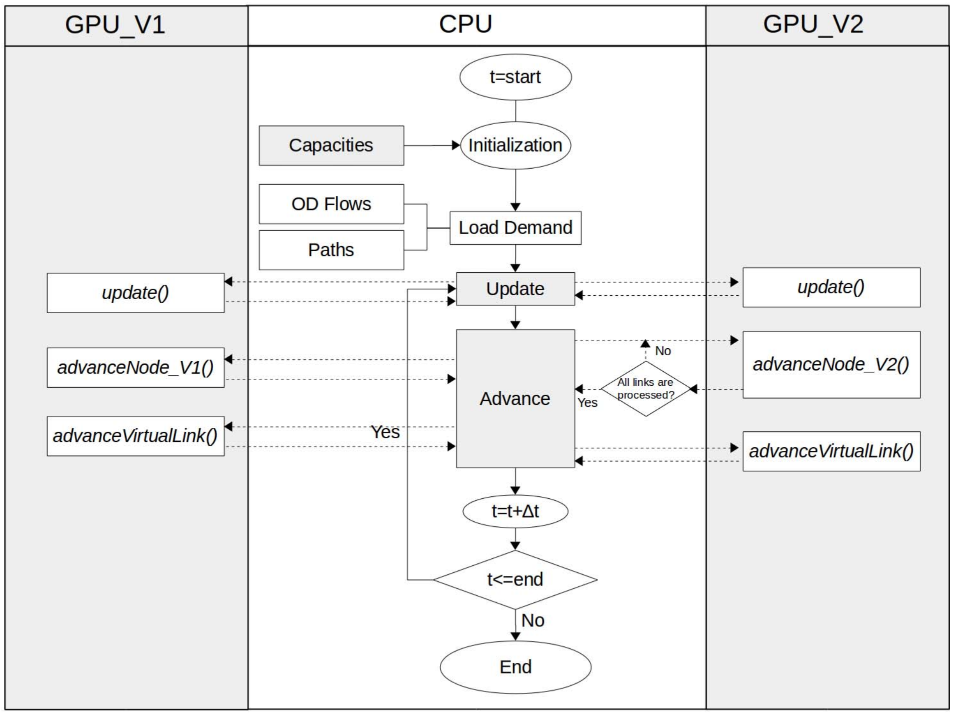

Fig. 2. Simulation Loop in Sequential and Parallel Traffic Simulations

plethora of information about GPU and its application could be found in many different sources. The review in [17] serves as a very good summary about GPU architecture, programming strategies and trends. Within the scope of this paper, it would be repetitive to discuss about GPU all over again but we would like to emphasize the following key points which have guided our design:

1) Architecturally, the CPU consists of several complex cores with complicated instruction-level parallelism and large caches and can handle few threads simultaneously. On the other hand, a GPU consists of hundreds of simple cores dedicated to computation which can handle thousands of threads at a time. With this difference in design, CPU is built to minimize latency while GPU is built to maximize throughput.
2) GPU executes instructions in a *Single Instruction Multiple Data (SIMD)* fashion, in which the same instruction is simultaneously executed for different data elements, called a warp. With this in mind, the application which benefits from GPU the most is the one with high level of data parallelism.
3) Threads in GPU are grouped into blocks which again are put in a grid. Threads can access different memory with different latency ranging from fast per-thread private memory to per-block shared memory and global memory. GPU hides memory access latency by multi-threaded architecture. It is also desirable to have *memory coalescing* when multiple threads access contiguous memory locations leading to fewer memory transactions and increased bandwidth.

With the above background information about mesoscopic traffic simulation and GPU, the question at hand is how to fit the modeling principles of mesoscopic simulation with high level of data dependency into GPU architecture to gain performance speed up. In the next section, the answer for this question will be discussed in detail.

III. A Framework for Mesoscopic Traffic Simulation in GPU

A. The Simulation Loop

A typical simulation of traffic network operations is illustrated in the middle column of Figure 2. Basically, a time-step mesoscopic traffic model proceeds in two phases: the update phase and the advance phase.

1) Update Phase: Update phase updates traffic dynamics at the link level (density, speed, queue, etc.) which are used in the current interval.

2) Advance Phase: Advance phase advances vehicles along their selected path to the new positions at the end of the simulation interval. The time step is set to a small enough value so that vehicle can just move along a link or is transfered from one link to the next. Vehicle cannot traverse more than two links within one simulation time-step. When a vehicle moves along the link where it is currently located at, the vehicle movement follows traffic dynamics calculated in the update phase. However, when a vehicle is in the queuing part at the end of a link, it can only move to next link if three conditions are fulfilled:

1) The current link has enough output capacity (which is updated by downstream end node)

2) The next link has enough input capacity (which is updated by upstream end node)
3) The next link has available empty space (which is updated by both upstream and downstream end nodes).

The first two conditions are local to each node, however the third one creates upstream-downstream data dependency because the available empty space of each link depends on both the upstream and the downstream nodes.

B. Sequential Version

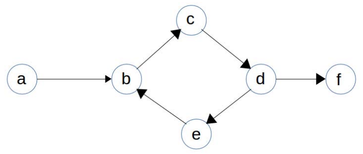

Fig. 3. Sample Network with Cycle

The sequential implementation of mesoscopic simulation loop is straight-forward. For each time step, update and advance phases are executed on each link one by one. Sequential version resolves upstream-downstream dependency by processing links in downstream-upstream order. This guarantees that a vehicle only moves from one link to another when the leading link has already been processed and the availability of empty space in the leading link has been confirmed. But there is an exception when the network contains cycles. Figure 3 shows a sample network with a cycle between 4 nodes *b, c, d and e*. After the link between node *d* and node *f* is processed, next link cannot be selected since the remaining links wait for each other in a cycle. To handle the situation when a vehicle moves to a link which has not been processed, one method called virtual processing was proposed [18]. As illustrated in Figure 4, vehicles moving through Node *a* which have not been processed are stored temporarily in a virtual link associated with Link A. Once Link A is processed, vehicles on this virtual link can be moved. The capacity of virtual link associated with Link A is set to the same as input capacity of Link A.

With virtual processing in place, advance phase could be split into two smaller steps. In the first step, links are processed sequentially, vehicles are moved along the links or from one link to another. When upstream-downstream dependency happens, virtual processing is employed to move vehicles to virtual links. In the second step, virtual links are processed sequentially to move vehicles on virtual links to actual link based on the available space of actual link. For both two steps, vehicles in each link are processed in downstream-upstream order.

Virtual processing contains an assumption about empty space of links since at the time the vehicle makes a move, the downstream link, which the vehicle is trying to move to, has not been processed and there is no way of determining the number of vehicles that link could take. There is a chance that there are more vehicles in virtual link than what the

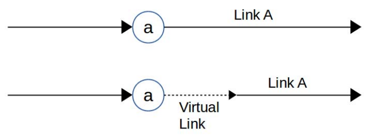

Fig. 4. Virtual Processing

actual link could handle which leads to vehicles getting stuck in virtual link till future time steps. This assumption creates the gap between real life traffic network operation and traffic simulation. In order to minimize virtual processing, sequential simulation sorts links in downstream-upstream order to ensure that downstream links are processed before vehicles from upstream links are transferred. Virtual processing is still needed to break cycles when necessary.

In the next sections, we will propose two different versions of GPU implementation to handle mesoscopic traffic simulation. One version compromises the simulation correctness in exchange for higher level of parallelism and the other version preserves simulation correctness with extra cost of synchronization.

C. Parallel Version 1: Virtual Processing All Links

In order to fit the simulation loop into GPU architecture and gain performance speed-up, it is crucial that data parallelism could be derived within the simulation. Let us have a closer look at the update and the advance phases.

1) Update Phase: The update phase updates traffic dynamics at the link level. Moreover, the to-be-updated traffic dynamics such as speed, density, queue, etc. only depend on the local state and characteristics of each link. It means the update of one link is completely independent of other links and multiple links could be updated in parallel. For this reason, it is straightforward to design a kernel function called *update()* to run update for network links, each link is processed on one GPU thread.

2) Advance Phase: Parallelizing the advance phase is more challenging due to the inherent downstream-upstream dependency.

As discussed in Section 3-B, CPU implementation of mesoscopic traffic simulation makes use of virtual processing to resolve downstream-upstream dependency when network cycle is encountered. CPU implementation also minimizes virtual processing by ordering links in downstream-upstream order. Borrowing the same technique, we design a GPU algorithm to conduct advance phase in parallel in which all links are virtually processed regardless of whether the downstream links have been processed. It means each vehicle's move is broken into two steps: first step is from current link to virtual link and second step is from virtual link to the downstream link.

Corresponding to the above two steps, we design two kernels: *advanceNode_v1()* and *advanceVirtualLink()*. The first kernel *advanceNode_v1()* moves vehicles from one link to the virtual link associated with the next link. *advanceNode_v1()* processes one node per one GPU thread. Since the input capacity and the output capacity of a link are updated by only one node, the first two conditions are resolved. Regarding the empty space of the next link, the virtual link helps store vehicles when the next link has not been processed and eliminate third condition. As a result, each GPU thread could run *advanceNode_v1()* for one network node independently.

In the second kernel *advanceVirtualLink()*, vehicles are moved from the virtual link to the actual link based on the available space of the actual link. Since the movement of vehicles from the virtual link to the actual link only involves one network link, each GPU thread could run *advanceVirtualLink()* for one network link independently. Algorithm 1 shows the pseudocode for the advance phase. The left column of Figure 2 shows the complete simulation loop and the associated GPU kernels to handle it.

Algorithm 1: Advance Phase with Virtual Processing

```
1  Function (advanceNode_v1())
2  for each vehicle on upstream links with time_to_leave ≤ current_time do
3      get next_virtual_link for vehicle
4      if current_link has output capacity AND next virtual_link has input capacity then
5          Remove from current_link
6          Move to next virtual_link
7      else
8          return ;        // no more vehicles can leave
9      end
10 end
11 Function (advanceVirtualLink())
12 for each vehicle on virtual link do
13     if actual_link has empty space then
14         Remove from virtual_link
15         Move to actual_link
16     else
17         return ;        // no more vehicles can leave
18     end
19 end
```

This version of advance phase has high level of parallelism since large network has thousands of nodes and links and they can be processed in parallel. However, a weakness of this method is that it relies on virtual processing for every link and may create incorrectness in the simulation results compared to the sequential simulation. However, uncertainties always exist in traffic simulation because a model is only an abstraction of the real network. It is difficult to conclude one model is correct or incorrect and there is not necessarily only one single correct model to one real network. Hence, this version of the advance phase is still useful if we can prove that the simulation error are statistically acceptable. This will be addressed in the experimental section.

D. Parallel Version 2: Preserving Simulation Correctness

This version of GPU-based mesoscopic traffic simulation has the same logic for the Update phase but the Advance phase is different. In the first version, *advanceNode_v1()* kernel needs to do virtual processing for all the network links and compromises simulation correctness but in this version, we propose a second, more refined *advanceNode_v2()* kernel to preserve the results of sequential simulation.

Our algorithm for the Advance phase has two main components:

- A subroutine to detect nodes for virtual processing
- *advanceNode_v2()* kernel with atomic operations

1) Detect network nodes for virtual processing: In a network with cycles, virtual processing is unavoidable even with sequential simulation. Therefore, a subroutine is designed to detect nodes for virtual processing. This subroutine is run offline only once for each network and the running time is not added to the total simulation time. The main ideas are shown in Algorithm 2.

Algorithm 2: Algorithm to mark network nodes which need virtual processing

```
1  Function (findNodesForVirtualProcessing())
2  repeat
3      for all nodes in the network do
4          if no downstream links then
5              remove the node and all its upstream links
6          end
7      end
8      if no node can be removed then
9          randomly select one node
10         mark the selected node for virtual processing
11         remove the node and all it upstream links
12     end
13 until Network is empty;
```

When Algorithm 2 runs for the sample network in Figure 3, first, Node *f* can be removed. After that, no other node is without downstream link so Node *d* is randomly marked for virtual processing. After Node *d* is removed, it is possible to further remove Nodes *c b e a* till the network becomes empty and the algorithm terminates. As a result, during simulation, the network in Figure 3 only needs to do virtual processing for Node *d*.

2) advanceNode_v2() kernel: This kernel is designed with the basic observation that: *as long as there is available empty space in the next link then the vehicle can move without waiting for the next link to finish processing*. This idea helps to increase the level of parallelism in the simulation.

In order to check if the next link has empty space and acquire it in one single operation (because another thread may update it in between checking and acquisition operations), we use atomic operation *atomicAcquire()*. This atomic operation checks if the next link has empty space. If yes, it decreases empty space by one vehicle length and stores the updated value back in the memory. To implement this atomic operation, we made use of CUDA *atomicCAS(int* address, int compare, int val)* routine which reads a 32-bit or 64-bit word *old* located at the address *address* in global or shared memory, computes *(old == compare ? val : old)* , and stores the result back to memory at the same address. These three operations are performed in one atomic transaction.

Regarding the performance of atomic operation in GPU, it is promising that the throughput of global memory atomic operations on new GPU architectures (Kepler, Maxwell) is substantially improved. In Kepler, atomic operation throughput to a global addresses is significantly accelerated, and logic to handle address conflicts has been made more efficient. Atomic operations can often be processed at rates similar to global load operations [19].

Algorithm 3: Algorithm to advance vehicles with preserved correctness

```
1  Function (function advanceNode_v2())
2  foreach vehicle upstream with time_to_leave ≤ current_time do
3      if vehicle reaches destination then
4          remove vehicle from the network
5          update current_link
6      else
7          if current_link.output_capacity > 0 AND next_link.input_capacity > 0 then
8              if next_link is processed then
9                  if next_link.empty_space > 0 then
10                     move vehicle to next_link
11                 else
12                     mark current_link as processed
13                 end
14             else if node is marked for virtual processing then
15                 move vehicle to virtual_link
16             else if atomicAcquire(next_link.empty_space) then
17                 move vehicle to next_link
18             else
19                 try again in next iteration
20             end
21         else
22             mark current_link as processed
23         end
24     end
25 end
```

The whole logic of *advanceNode_v2()* kernel is illustrated in Algorithm 3. In order to implement this logic in GPU with CUDA, it is important to understand the following GPU architecture and CUDA scheduling principles:

- CUDA does not support global thread synchronization unless kernel is terminated and returns to CPU. Some GPU studies proposed global thread barrier for global synchronization [20] but all with added cost.
- In GPU, threads from a block are bundled into fixed-size warps (usually size 32) for execution on a CUDA core, and all threads in the same warp must execute the same instruction at the same time. In other words, threads cannot diverge. If branching exists in kernel code, threads in different branches will be executed in sequence. It means, some threads may need to wait for other threads to finish their execution before getting allocated computing resources and the execution order cannot be guaranteed. Thread divergence can cause deadlock especially in code with a lot of dependencies between threads like the proposed *advanceNode_v2()* kernel.

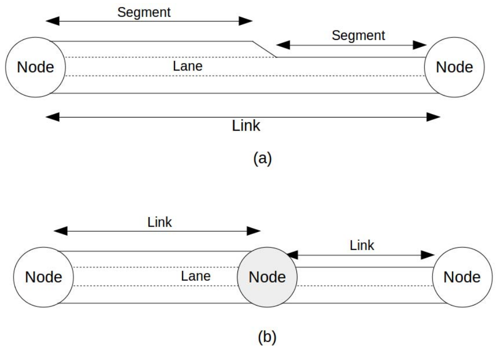

Fig. 5. (a)Network Node-Link-Segment-Lane Structure (b)Transformed Network

From the above observations, the proposed algorithm has been designed in such a way that each *avanceNode_v2()* kernel only moves one vehicle a time and CPU repeatedly calls *advanceNode_v2()* kernel until all the nodes are processed. It means, threads are synchronized by returning control to CPU. This implementation incurs extra overhead due to multiple kernel launches but saves the program from hitting a deadlock due to un-manipulable GPU scheduling. The right column of Figure 2 shows the flow of second GPU-based version.

The proposed *advanceNode_v2()* algorithm minimizes virtual processing and follows exactly the same logic of sequential simulation. However, it has two drawbacks. First, it has lower level of parallelism than the first version due to the fact that not all nodes can pass the three conditions check to move the vehicles. Second, extra synchronization cost is added into total simulation time.

IV. Implementation

A. Network Transformation

In the previous sections, traffic simulation was discussed in perspective of links and nodes. However, the input traffic network is more structured than that. Each link in the network is divided into segments that capture variations in traffic conditions along the link. Each segment contains a set of lanes which capture the queuing behaviour of a vehicle within the segment. Vehicles are assigned to lanes based on their paths. Figure 5(a) shows an example of one link in the network with 2 segments, one has 3 lanes and the other has 2 lanes.

Vehicle movement in the network follows traffic dynamics and capacity constraints of segments while adhering to queuing processes of lanes. In sequential simulation, links

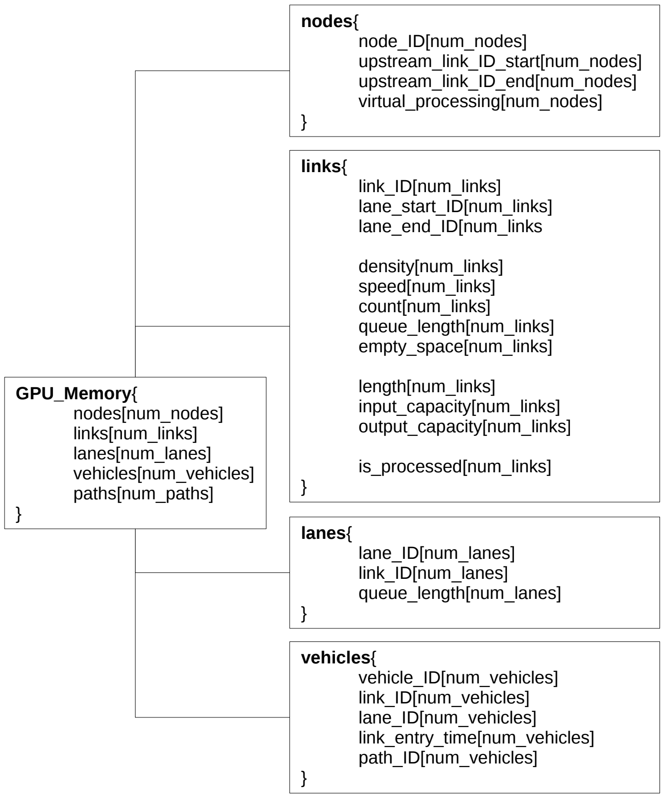

Fig. 6. Data Structure for Network Components and Vehicles

are processed one by one and within each link, segments are processed in downstream to upstream order to ensure realistic movement of vehicles. From this fact, we transform the network as follows:

- Add virtual nodes between segments
- Convert segments into links

When GPU algorithm is applied into the transformed network, the kernel function *update()* operating on each link needs extra logic to handle lanes and lane queues. Kernel functions *advanceNode_v1()* and *advanceNode_v2()* need extra logic to handle lane selection (vehicle chooses lane with shortest queue to travel). The kernel function *advanceVirtual()* now operates at virtual lane level. It means, each lane has one associated virtual lane to store temporary vehicle attempting to move to this lane. The general principle of mesoscopic simulation logic is still guaranteed and the level of data parallelism is increased due to increased number of nodes and links.

B. Optimal Data Structures

As proven in previous studies [8], [10], when data is organized in structures of arrays, two adjacent kernels will have coalesced memory accesses. Applying this principle to our framework, containers for vehicles and road network components are designed as structures of arrays. The simplified structure of data in GPU global memory is illustrated in Figure 6.

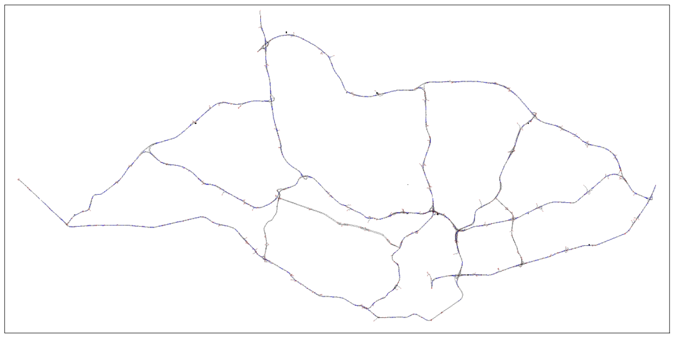

Fig. 7. The Singapore Expressway Network

V. Evaluation

A. Experiment Setup

1) Network: To evaluate the proposed framework, a testbed is set up for an actual network of Singapore Expressways (Figure 7) which has 845 nodes and 1040 links. After transformation, the updated network consists of 3186 nodes, 3386 links and 9437 lanes. Since the number of nodes, links and lanes define the number of device threads to execute the kernels in parallel, network transformation did increase the level of data parallelism.

2) Demand: Vehicle trips follow 4103 origin-destination pairs for a simulation period between 06:30 AM and 08:00 AM (30 minutes warm-up period). The simulation time step is set to a minimum value (1 second) to further ensure simulation correctness.

Three levels of network demand: Low (100k vehicles), Medium (200k vehicles) and High (300K vehicles) are tested. For each level of demand, 20 sets of vehicle trips are generated with different random seeds. Results of all data sets are averaged to produce final output for comparison. This further ensures that our experiments cover different network conditions.

3) Benchmarking: Three versions of mesoscopic traffic simulation are implemented according to algorithms discussed in Section 2. The first version follows logic of sequential simulation in CPU, the second version moves vehicles on network based on virtual processing principle (GPU_V1) and the third one makes use of atomic operations and synchronization to resolve upstream-downstream data dependency (GPU_V2). GPU_V1 and GPU_V2 are benchmarked against CPU version to analyze performance speedup.

4) Platforms: Experiments were conducted on a consumer PC. The CPU is Intel i7-4980HQ (2.80GHz) with 4 cores, 16GB memory and 1TB SSD Hard Disk. The GPU is GeForce GT 750M with 2 multiprocessors, 384 cores and 2GB global memory. GPU codes are implemented with CUDA 8.0 [21]. Both CPU and GPU codes are run on Ubuntu 14.04LTS.

B. Results

1) Performance Speedup: Figure 8 illustrates the experimental results. Each line in this figure corresponds to the

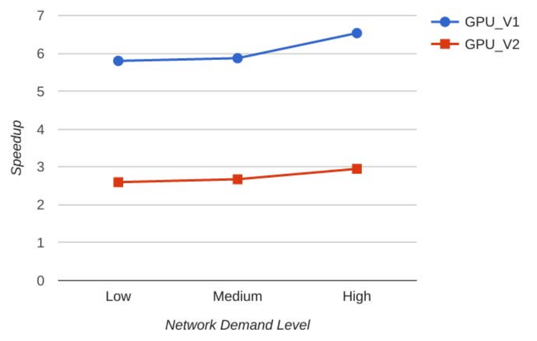

Fig. 8. Speed up of GPU versions compared to CPU version

speedup of one GPU-based version compared to CPU version. As clearly shown in the figure, GPU out-performs CPU considerably in speeding up the simulation with speedup of version 1 ranging from 5.8 times to 6.5 times and speed-up of version 2 ranging from 2.6 times to 3 times with different demand levels.

Another observation is that when demand increases, performance gain by switching to GPU is better. This further shows the resilient of GPU in handling heavy workload.

Moreover, the results also confirm that GPU_V1 has better performance speedup than GPU_V2 as we expected. The reason comes from the fact that GPU_V1 exchanges simulation correctness for higher level of parallelism by simulating all network links using virtual processing while GPU_V2 adheres to the same logic of sequential simulation with extra cost of synchronization and atomic operations.

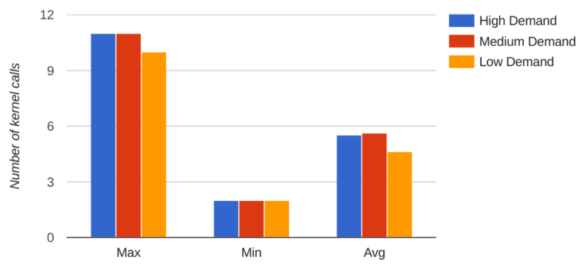

Fig. 9. MAX, MIN avd AVG number of advanceNode_V2() kernel calls in GPU_V2 version

To further understand why GPU_V2 has exactly the same logic as sequential simulation but could achieve up to 3 times speedup, we plot another chart in Figure 9 to get more insight into number of advanceNode_V2() kernel calls that GPU_V2 needs to make in order to advance vehicles during simulation. This figure shows the maximum, minimum and average number of kernel calls over all simulation time steps. As seen in Figure 9, with different demand levels, the maximum number of advanceNode_V2() kernel launches is 11, the minimum is 2 and the average is about 5. Compared to sequentially processing more than 3000 nodes, GPU_V2 achieves much higher level of parallelism while still maintaining the simulation correctness.

This result implies that with good design and algorithms, GPU could outperform CPU with 4 cores and good configuration. With reasonable cost compared to complicated clusters, GPU would be a promising candidate in speeding up large scale applications.

2) Simulation Correctness: Since the GPU_V2 has the same logic as the sequential CPU simulation, it needs no further check for simulation correctness. In this section, we will investigate the output of GPU_V1 with virtual processing and compare it against the output of sequential simulation.

In mesoscopic traffic simulation, the macro traffic dynamics such as speed, density, count are usually important outputs. For this reason, we compare output segment speeds, densities and counts of GPU_V1 against the ones of CPU version. For each data set, each version produces three output vectors of segment speeds, densities and counts. The Root Mean Square Errors (RMSE) between output vectors of GPU_V1 version and CPU version are then calculated according to the following formula:

$$RMSE = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(s_i - p_i)^2} \tag{1}$$

in which, s_i is output of sequential simulation, p_i is output of parallel simulation. N is number of values in the output vector.

RMSE is a very common error metric which amplifies and severely punishes large errors so it is a good candidate to quantify the differences in two vectors of simulation output. Since we have 3 levels of demand, each has 20 data sets, each data set produces 3 output vectors corresponding to speeds, densities and counts, we will eventually have 9 combinations of demand level (low-medium-high) and traffic property (speed-density-count), each combination has 20 values of RMSE.

To estimate the range of RMSE, we makes use of the *Boostrap* method [22] which is very useful when the distribution of the tested variables are not known. The core idea of Bootstrap is that it uses a small sample data set and conducts resampling-with-replacement a large number of times to perform inference about the sample data from the resampled data. In our case, each combination of demand level and traffic property has 20 values of RMSE. The Bootstrap method conducts resampling-with-replacement on this set of RMSE 1000 times and produces 1000 new data sets. Mean RMSE is then calculated for each data set. From here, a distribution of mean RMSE and the estimated range of RMSE can be derived.

Table I lists out the range of RMSE for 9 combinations of demand level and traffic property using Bootstrap method with 95 % confidence interval. In our opinion, RMSE values between CPU output and GPU_V1 output are within acceptable ranges for a large network like Singapore expressway network. However, this test can be repeated for any other networks with different criteria to judge if the performance gained from virtual processing is worth the compromised correctness.

TABLE I
RMSE RANGES

	Speed (km/h)	Density (veh/km)	Count (veh)
Low Demand	1.505-1.595	4.8-5.2	2.065-2.242
Medium Demand	1.422-1.534	4.2-4.8	1.567-1.858
High Demand	1.429-1.548	4.2-4.6	1.613-1.853

VI. CONCLUSION

This paper has proposed a framework to implement mesoscopic traffic simulation in GPU. The proposed framework has high level of data parallelism and optimal data structure which suits GPU architecture and provides significant speedup compared to the CPU implementation. Two versions of GPU-based simulation with different levels of correctness have also been discussed with reported results and in-depth analysis. Through the work derived in this paper, three important key points are observed:

- In order to utilize GPU's power, it is very important that there exists high level of data parallelism in the application.
- When working with application with high level of data dependency such as simulation, in exchange of data parallelism and performance, some correctness may be lost. The tolerance for correctness loss needs to be weighed-in during design phase.
- The choice of data structure plays a very important role in optimizing GPU code. Good data structure encourages memory coalesced accesses and minimizes memory operations which are expensive in GPU.

GPU architecture and performance are being updated and enhanced everyday. The opportunities are vast and promising. This work will definitely be extended in the following directions:

- On the simulation side, this work currently focuses on the supply side while the demand side is simplified. For future work, the demand side will get a better treatment with consideration of driver behaviours. Moreover, other features (such as incident management, traffic prediction, strategy optimization) will also be implemented in GPU.
- On the GPU side, more optimization will be done at a deeper level of GPU architecture regarding warp execution efficiency. Even though the current data structure has reduced memory access efficiently, a lot of uncoalesced memory accesses still exist due to the hierarchical structure of network components. Further optimization of memory access will definitely bring more speedup.

ACKNOWLEDGMENT

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).

REFERENCES

[1] J. Barcelo, *Fundamentals of Traffic Simulation*, vol. 145 of *International Series in Operations Research & Management Science*. New York, NY: Springer, 2010.

[2] M. Ben-Akiva, H. N. Koutsopoulos, C. Antoniou, and R. Balakrishna, *Fundamentals of Traffic Simulation*, ch. 10-Traffic Simulation with DynaMIT. International Series in Operations Research & Management Science, 2010.

[3] M. Ben-Akiva, H. N. Koutsopoulos, C. Antoniou, and R. Balakrishna, *Fundamentals of Traffic Simulation*, ch. 5-Traffic Simulation with Aimsun. International Series in Operations Research & Management Science, 2010.

[4] H. Aydt, Y. Xu, M. Lees, and A. Knoll, "A multi-threaded execution model for the agent-based semsim traffic simulation," in *Proceedings of 13th International Conference on Systems Simulation* (G. T. et al., ed.), vol. 402, (Singapore), pp. 1–12, 2013.

[5] R. Klefstad, Y. Zhang, M. Lai, R. Jayakrishnan, and R. Lavanya, "A distributed, scalable, and synchronized framework for large-scale microscopic traffic simulation," in *Proceedings of the 8th International IEEE Conference on Intelligent Transportation Systems*, (Vienna, Austria), pp. 813–818, 2005.

[6] "General-purpose computation on graphics hardware," 2017.

[7] P. K. S., "Efficient execution on gpus of field- based vehicular mobility models," in *Proceedings of the 22nd Workshop on Principles of Advanced and Distributed Simulation PADS*, (Roma, Italy), p. 154, 2008.

[8] D. Strippgen, , and K. Nagel, "Multi-agent traffic simulation with CUDA," in *Proceedings of International Conference on High Performance Computing & Simulation*, (Leipzig), pp. 106–114, 2009.

[9] Z. Shen, K. Wang, and F. Zhu, "Agent-based traffic simulation and traffic signal timing optimization with gpu," in *Proceedings of 14th International IEEE Conference on Intelligent Transportation Systems*, (Washington, DC, USA), pp. 145–150, 2011.

[10] Y. Xu, G. Tan, X. Li, and X. Song, "Mesoscopic Traffic Simulation on CPU/GPU," in *Proceedings of the 2nd ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, (Colorado, USA), pp. 39–50, 2014.

[11] X. Song, Z. Xie, Y. Xu, G. Tan, W. Tang, J. Bi, and X. Li, "Supporting real-world network-oriented mesoscopic traffic simulation on GPU," *Simulation Modelling Practice and Theory*, vol. 74, pp. 46–63, 2017.

[12] Y. Sano, Y. Kadono, and N. Fukuta, *A Performance Optimization Support Framework for GPU-Based Traffic Simulations with Negotiating Agents*, vol. 638 of *Recent Advances in Agent-based Complex Automated Negotiation Studies in Computational Intelligence*. 2016.

[13] D. P. Leonard, P. Gower, and L. Taylor, "CONTRAM. Structure of the Model," Transport and Road Research Laboratory, Research Report 178, Department of Transport, Crowthorne, 1989.

[14] R. Jayakrisham, H. S. Mahmassani, and T. Y. Yu, "An evaluation tool for advanced traffic information and management systems in urban networks," *Transport Res C*, vol. 2C, no. 3, pp. 129–147, 1994.

[15] W. Yang, *Scalability of Dynamic Traffic Assignment*. PhD thesis, Massachusetts Institute of Technology, Massachusetts, MA, 2009.

[16] J. Cheng, M. Grossman, and T. Y. McKercher, *Professional CUDA C Programming*. Indiana: John Wiley & Sons, Inc., 2014.

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

[18] S. Rajagopal, "Development of a Mesoscopic Traffic Simulator for DYNAMIT-Dynamic Network Assignment for the Management of Information," Master's thesis, Carnegie Mellon University, 1998.

[19] NVIDIA Corporation, "NVIDIA's Next Generation CUDA Compute Architecture: Kepler TM GK110," 2012.

[20] S. Xiao and W. Feng, "Inter-block GPU communication via fast barrier synchronization," in *Proceedings of the IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, (Atlanta, GA), pp. 1–12, 2010.

[21] NVIDIA Corporation, "CUDA C Programming Guide," 2016.

[22] B. Efron and R. Tibshirani, "Bootstrap Methods for Standard Errors, Confidence Intervals, and Other Measures of Statistical Accuracy," *Statistical Science*, vol. 1, no. 1, pp. 54–75, 1986.