Acceleration of Agent-Based Traffic Flow Simulation with Numba CUDA

Zuzhao Ye

***Abstract*— Agent-based traffic simulation is widely used in traffic forecasting. Recent years, traffic models are getting more complex and the requirement of precision also increases. We see challenges in the computation side and the necessity of acceleration. In this article, we formulated an agent-based traffic model. Then we developed a naive algorithm and an optimized algorithm for this model. Both serial and parallel versions of these algorithms are implemented with the support of Numba CUDA, a parallelizing library for Python. We achieved around 10 times acceleration with our optimized parallel algorithm.**

I. Introduction

In transportation engineering, forecasting the traffic flow and traffic density is of critical importance for transportation management as well as new infrastructure design. Traditional field measurement is a resource-intensive task, which relies on human observation or the installation of massive sensors, e.g. magnetic detectors. Recent development of computer vision enables efficient traffic measurement based on camera. Nevertheless, computer vision can only measure existing traffic and not able to forecasting the future conditions.

To address the forecasting problem, Nagel and Schreckenberg [1] proposed an agent-based model for single-lane traffic simulation, which is called NaSch model. In NaSch model, a road section is divided into cells of certain length. Each vehicle on the road will occupy a cell and their velocities and positions will be updated over each time step by following micro-scale traffic rules, e.g. vehicle spacing, reaction time, and speed limit. Through iterations, the system is expected to reach an equilibrium state, from which we can read the relationship between traffic flow and density.

Over years of development, more elements have been added to the NaSch model, e.g. lane changing, intersection, and etc. At the same time, the size of cells is getting smaller for more precise results, and the length of road section is increasing to support larger traffic networks. These bring challenges to the computation side. Using a single-threaded CPU cannot meet the drastically increased demand.

To address the limitation of CPU, [3] introduces a GPU accelerated traffic flow simulation on a single-lane road. Further, [4] also used GPU to accelerate traffic simulation with intersection signals. Thinking more broadly, the agent-based simulation is not limited to traffic, it is widely adopted in other fields and also need acceleration. For example, [2] introduces several parallel algorithms for agent-based simulation in the field of biology.

[1]Zuzhao Ye is a graduate student of Electrical Engineering, University of California, Riverside, CA, USA zye066 at ucr.edu

More recently, [5] brings us GEMSim, a platform for large scale city-wide parallel simulations.We also have MASS CUDA [7], a parallelizing library for multi-agent spatial simulation, designed to support a wide range of applications. Beyond GPU, [6] also tries to use FPGA to parallelize and accelerate agent-based simulations.

Given such an active field, we are particularly interested in developing our own parallel algorithms for two reasons: 1) To get a sense of how to parallelize agent-based simulations. 2) Get prepared for future career and research.

This article will be divided into several sections. Section II formally formute the problem. Section III analyzes the challenges in computation. Section IV introduces both serial and parallel algorithms. Section V performs case study on these algorithms. Section VI summarizes the achievements of this paper and potential future improvements.

II. Problem Formulation

A. Purposes

For a traffic simulation, we are interested in obtaining macro parameters through micro traffic rules which govern the interaction between individual vehicles. Typical macro traffic parameters include traffic flow Q, density ρ, and average speed v_m. They are connected through Eq. (1).

$$Q = \rho v_m \tag{1}$$

The basic idea is, in a self-connected road section with a certain number of vehicles, the system can reach an equilibrium state over time through interactions between individual vehicles. Fig 1 illustrates such a system. The vehicles are given unique IDs. They will travel from left to right, then moving back to the left side. Their position and speed will be initialized randomly. Over a certain steps of interaction, the system will reach an equilibrium state, in which the average speed v_m converges to a certain value. Since no vehicle can escape from the system, the total number of vehicles N_v will remain the same. Therefore, we can obtain traffic density ρ and average speed v_m through Eq. (2) and (3):

$$\rho = \frac{N_v}{L_r} \tag{2}$$

$$v_m = \sum_{i=1}^{N_v} v_i \tag{3}$$

Where L_r is the length of the road section, and v_i is the speed of vehicle i.

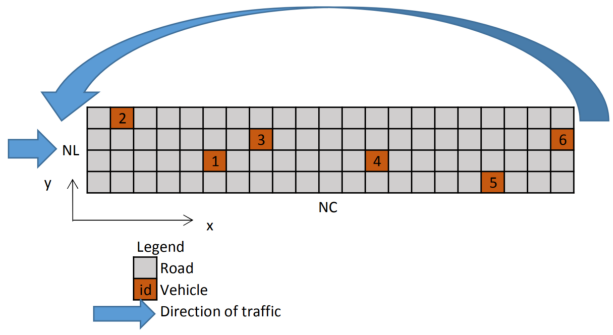

Fig. 1: Demo of a road section, vehicle, and direction of traffic (N_L: Number of lanes, N_C: Number of cells on a lane)

B. Micro-scale Traffic Rules

Now we introduce the micro traffic rules, which are mathematical expressions of our everyday driving rules. For a straight road section, there are two major rules: driving forward and change lane.

1) Rule of Driving Forward: First of all, we consider the safe distance d_s between vehicles, which is determined by the speed of the rear vehicle v_r and the reaction time of the driver t_r:

$$d_s = v_r t_r \tag{4}$$

A vehicle's acceleration a will generally related its distance to the front vehicle d_f and the safe distance d_s:

$$a = \alpha(d_f - d_s) \tag{5}$$

When $d_f > d_s$, the vehicle tends to accelerate, and when $d_f < d_s$, the vehicle will decelerate. α is an coefficient. An example is show in Fig 2 (top), where Veh 1 can accelerate because Veh 4 is further than d_s. Veh 3 can do max acceleration $a_{max} = \alpha(d_{f,max} - d_s)$ because there is no vehicle in front of it within the checking range.

During the transition from time step t to $t+1$, we will update the postion x and speed v of a vehicle by following Eq. (6) and (7):

$$v_{t+1} = v_t + a\Delta t \tag{6}$$

$$x_{t+1} = x_t + v\Delta t \tag{7}$$

2) Rule of Lane Changing: Similar to driving forward, lane changing also requires safety checking, but it checks cells in the adjacent lanes instead of cells in the same lane, as illustrated in Fig. 2 (bottom). In Fig 2 (bottom), Veh 3 can switch to its left lane, because there is no vehicle within the range of safety distance (for both rear and front direction). It is not safe for Veh 3 to switch to its right lane, because Veh 1 is too close. Eq. (8) is a general expression of safe condition S regarding lane changing, where $\beta \in \{left\ (-1), right\ (+1)\}$, d_f and d_r are the distances to the front or rear vehicles.

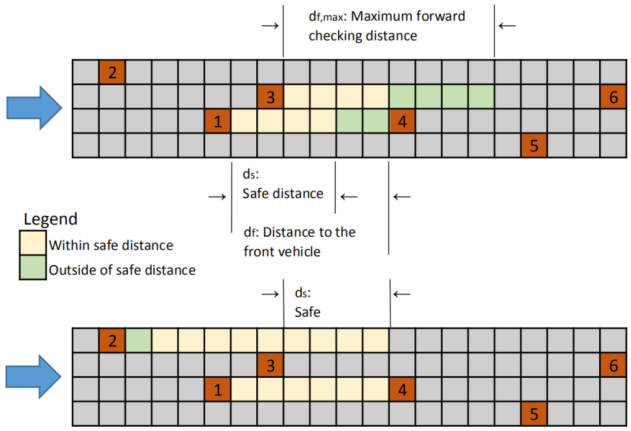

Fig. 2: Demo of safety checking for forward driving (top) and lane changing (bottom)

$$S_\beta = \begin{cases} True & \text{if } d_{f,\beta} \geq d_s \text{ and } d_{r,\beta} \geq d_s \\ False & \text{if } d_{f,\beta} < d_s \text{ or } d_{r,\beta} < d_s \end{cases} \tag{8}$$

In addition to safety, a vehicle also determines necessity before doing any lane changing. When its distance to the front vehicle d_f is smaller than their distance to the front vehicles in adjacent lanes $d_{f,\beta}$ by a certain amount, the necessity of lane changing N is triggered. This is expressed in Eq. (9).

$$N_\beta = \begin{cases} True & \text{if } d_f < d_{f,\beta} - C \\ False & \text{if } d_f \geq d_{f,\beta} - C \end{cases} \tag{9}$$

A lane changing to β direction is triggered if and only if $S_\beta \ \& \ N_\beta = True$, then the vehicle will switch to β direction under a certain probability P_c, as shown in Eq. (10), where y is the lane number.

$$y_{t+1} = \begin{cases} y_t + \beta & Prob = P_c \\ y_t & Prob = 1 - P_c \end{cases} \tag{10}$$

III. Computation Challenges

To simulate the system as illustrated in Section II, there are three major challenges in computation.

1) **Forward checking**. As shown in Fig 2, for each vehicle, we need to read a lot of cells in front of it to determine the acceleration a.

2) **Lane changing checking**. This requires approximately 4 times memory reads compare to the above forward checking, because all 4 possible directions need to be checked before the vehicle can switch lane.

3) **Small cell size**. Challenge 1) and 2) will become more severe if a smaller cell size is used. The original NaSch model divides each cell to represent 7.5m (24.6 ft) physical space. With such a cell size, it is impossible to achieve any

results more precise than 7.5m. For example, if we use a 1 second time step, the speed can only takes values 0m/s, 7.5m/s, or 15m/s ,and so on. To have better precision, the ideal cell size Δd is determined by considering the precision of speed we pursue as shown in Eq.(11):

$$\Delta d = \Delta v \Delta t \tag{11}$$

Where Δv is the minimum interval between two speed values. If we want a precision of 1m/s, and Δt is 0.5s (human reaction time), then the cells size Δd should be 0.5m. In the future, when we have more autonomous vehicles, which have much less reaction time, then we need an even smaller cell size. Fig.**??** illustrates the relationship between cell size and the total number of cells in a road section.

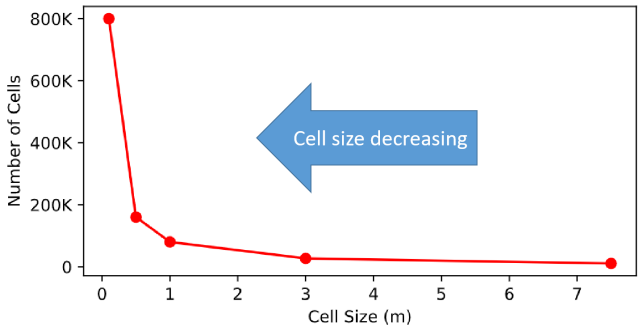

Fig. 3: Number of Cells vs. Cell Size for a road section with length 10km (6.2 miles), No. of Lanes 8)

IV. Implementation

A. Selection of Tools

Given the challenges as stated Section III, using serials CPU algorithm can be slow. We also notice that the update of a vehicle's properties at time step $t+1$ does not rely on other vehicles' updates at $t+1$, instead, it solely relies on other's information at t. This makes parallel algorithm a good choice.

We will use Python, a widely used prototyping language, in our implementation. Since Python is an interpreted language, it can be slow compared with other pre-compiled languages, e.g. C++, especially in For and $While$ loops. This makes Just-in-Time (JIT) compilation a necessity for Python. We have Numba, a Python library, enables JIT compilation for Python. Numba can compile any reuse-intense functions to machine code during the first run, then the entire program can run as fast as other pre-compiled languages. A comparison between execution time with and without JIT will be shown in Section xx.

What's more wonderful, Numba is integrated with CUDA, a parallel computing platform made for NVIDIA GPU. We can write Python style code for parallel computations. With JIT and CUDA altogether in one library, we can perform fair comparison between CPU and GPU algorithms.

B. A Naive Algorithm

We start from a straight forward algorithm: Loop over each cell in the road matrix, as illustrated in Algorithm 1.

Algorithm 1: A Naive CPU Algorithm

```
Initialize road matrix M_0;
Randomly generate vehicles on road matrix M_0;
while t < t_end do
    M_t' = M_t;
    M_{t+1} = M_t;
    for i ∈ M.rows do
        for j ∈ M.columns do
            if M_t(i, j) has a vehicle then
                Check lane changing according to Eq.
                  (8) and (9) based on M_t;
                Update vehicle's lane position y
                  according to Eq. (10) to M_t';
            end
        end
    end
    for i ∈ M'.rows do
        for j ∈ M'.columns do
            if M_t'(i, j) has a vehicle then
                Calculate acceleration a according to
                  Eq. (4) - (5) based on M_t';
                Update vehicle's speed v according to
                  Eq. (6);
                Update vehicle's travel position x
                  according to Eq. (7) to M_{t+1};
            end
        end
    end
    t = t + 1
end
```

Accordingly, we can develop an associate parallel algorithm, which uses 2D thread blocks to cover the road matrix. A thread will be activated when there is a vehicle in the cell. See Algorithm 2 for kernel functions and Algorithm 3 for the main function that integrates the kernels. The design of thread blocks are illustrated in Fig 4.

Algorithm 2 (and 3) has a major problem: warp divergence, because each thread needs to operate on a cell, regardless of whether this cell has a vehicle or not. We will introduce an optimized algorithm in the following section.

C. An Optimized Algorithm

An optimized algorithm is to use separate 1D matrices Pos_x and Pos_y to store vehicles' x and y position index. It benefits both CPU and CUDA programs. For CPU, previously we need two For loops on road matrix M to check each cell, now we can directly access their positions with one shorter For loop on $Pos_{x/y}$. For CUDA, previously we need to use 2D thread blocks to cover the entire road matrix M, now we only need 1D thread blocks to cover $Pos_{x/y}$

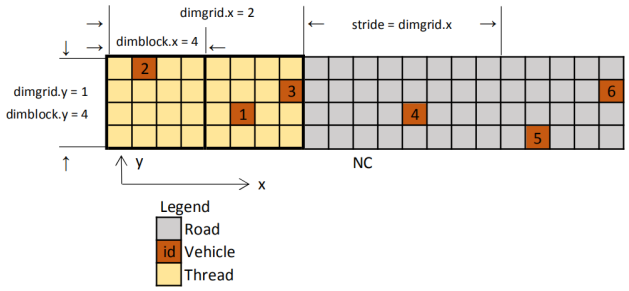

Fig. 4: Demo of the thread block of the naive CUDA algorithm

Algorithm 2: A Naive CUDA Algorithm (Kernels).

```
i = threadIdx.y;
j0 = threadIdx.x + blockIdx.x * blockDim.x;
stride = blockDim.x * gridDim.x;
if j0 < NC and i < NL then
    for j ∈ (j0, NC, stride) do
        Kernel 1: detect_lane_changing(M, R);
        if M(i, j) has a vehicle then
            Check lane changing Δi based on M;
            Record Δi to R;
        end
        Kernel 2: perform_lane_changing(M, R);
        if R(i, j) not zero then
            Δi = R(i, j);
            M(i + Δi, j) = M(i, j);
            M(i, j) = 0;
        end
        Kernel 3: detect_forward_driving(M, R);
        if M(i, j) has a vehicle then
            Calculate Δj based on M;
            Record Δj to R;
        end
        Kernel 4: perform_forward_driving(M, R);
        if R(i, j) not zero then
            Δj = R(i, j);
            M(i, j + Δj) = M(i, j);
            M(i, j) = 0;
        end
    end
end
```

Note that the boundary check will be launched for each kernel, but we use only one boundary check here because of the limited space.

(As shown in Fig 5), then there will be less overhead in launching blocks. More importantly, there will be less warp divergence, because each element in $Pos_{x/y}$ is guaranteed to corresponding to a vehicle.

Algorithm 3: A Naive CUDA Algorithm (Main)

```
Main;
Initialize road matrix M_0, recording matrix R;
Randomly generate vehicles on road matrix M_0;
Copy M_0 and R to device;
while t < t_end do
    detect_lane_changing[grid, block](M, R);
    perform_lane_changing[grid, block](M, R);
    detect_forward_driving[grid, block](M, R);
    perform_forward_driving[grid, block](M, R);
    t = t + 1;
end
Sync device and host;
Copy M to host;
Free M;
```

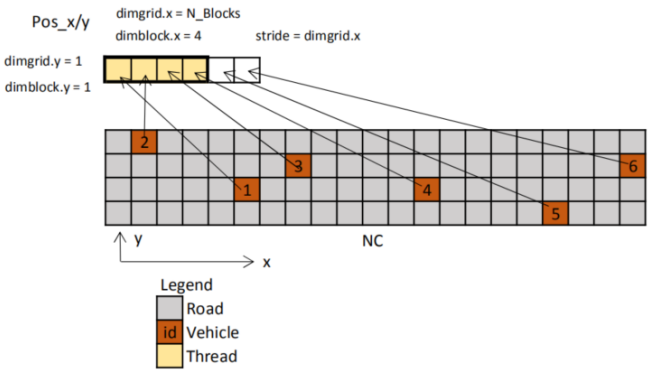

Fig. 5: Demo of the thread block of the optimized CUDA algorithm

Algorithm 4: An Optimized CPU Algorithm

```
...;
Get Pos_{x/y} from M_0;
while t < t_end do
    M'_t = M_t;
    M_{t+1} = M_t;
    for k ∈ N_v do
        i, j = Pos_y[k], Pos_x[k];
        Check and perform lane changing for
          M(i, j)...;
    end
    for k ∈ N_v do
        i, j = Pos_y[k], Pos_x[k];
        Check and perform forward driving for
          M(i, j)...;
    end
    t = t + 1
end
```

V. Case Study

In this section, we will compare the following items as discussed in previous sections.

Algorithm 5: An Optimized CUDA Algorithm (Kernels and Main)

```
k_0 = threadIdx.x;
stride = blockDim.x * gridDim.x;
for k ∈ (k_0, N_v, stride) do
    i, j = Pos_y[k], Pos_x[k];
    Kernel 1: detect_lane_changing(M, R);
    ...
    Kernel 2: perform_lane_changing(M, R);
    ...
    Kernel 3: detect_forward_driving(M, R);
    ...
    Kernel 4: perform_forward_driving(M, R);
    ...
end
Main;
...;
Get Pos_{x/y} from M_0;
Copy M_0, R, and Pos_{x/y} to device;
while t < t_end do
    ...;
    t = t + 1;
end
...;
```

- CPU cases with and without JIT.
- CPU and CUDA cases for naive algorithm (JIT is the default from now on).
- CPU and CUDA cases for optimized algorithm.
- CPU and CUDA cases for optimized algorithm with lane changing.

The hardware used are listed in Table I.

TABLE I: Hardware used in case studies

Item	Configuration
CPU	AMD Ryzen 7 2700 Eight-Core Processor
GPU	GeForce RTX 2080 Ti

The dependent libraries are listed in Table II

There are quite a few parameters that can impact the size of the problem as listed in Table III. For convenient of comparison, we will only change the road length, and fix all the rest.

We also define the parameters for CUDA as show in Table IV.

A. The Power of JIT

First of all, we compare the execution time of CPU cases with and without JIT. As shown in Table V, the program with JIT is 300x faster than that without JIT. This greatly demonstrate the power of JIT. In the rest of this paper, JIT will be the **default** setting for fair comparison between CPU and CUDA.

TABLE II: Libraries used in case studies

Item	Function
Python	Main programming language
Numba	Supports CUDA and JIT
Numpy	Create arrays and matrices

TABLE III: Model Parameters

Item	Value	Unit	Type
Road length L	0.1 - 100	km	Varied
Lane number	4	/	Fixed
Cell size	0.1	m	Fixed
Traffic density	30	vehicles/km/lane	Fixed
Total time steps (*)	100	/	Fixed

* It is found that the system usually converges to an equilibrium state within 100 time steps, see Appendix for further discussion.

B. Compare Naive and Optimized Algorithms

We are interested in how much faster is the optimized algorithm compared with the naive one. Fig 6 shows the simulation results for comparison between naive and optimized algorithms.

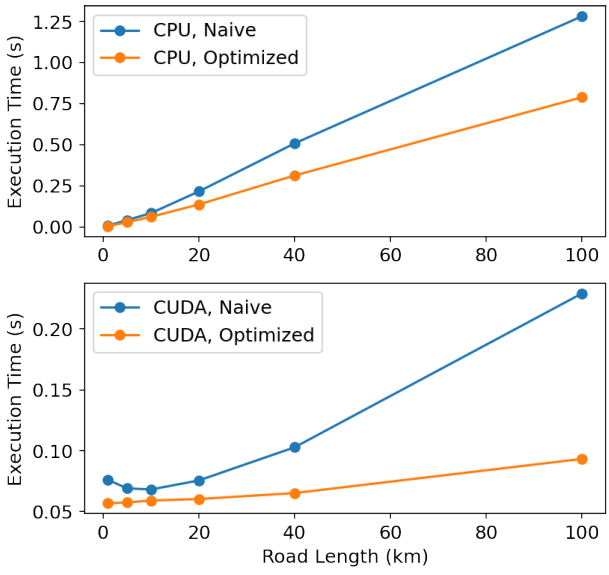

Fig. 6: Comparison of execution time between naive and optimized algorithms for CPU (top) and CUDA (bottom)

As we can see from Fig 6, the optimized algorithm is about 2x faster than the naive one for CPU, and about 3x

TABLE IV: CUDA Parameters

Algorithm	Grid size	Block size
Naive	2D, 32 × 1	32 × 4
Optimized	1D, 32	128 × 1

TABLE V: Comparison of Execution Time for CPU Cases w/ and w/o JIT

Time \ Config / Case	w/o JIT	w/ JIT	JIT Speedup
L = 0.1 km	50.3 ms	124 μs	$\sim$ 400
L = 1 km	503 ms	1.12 ms	$\sim$ 450
L = 10 km	2.79 s	9.2 ms	$\sim$ 300

* Total simulation time steps is 10 here instead of 100 for shorter waiting time.

faster for CUDA. Both CPU and CUDA benefit from the optimized algorithm, while for the CPU is the reduced *For* loops that matters, for CUDA it is less warp divergence that matters here.

C. Compare CPU and CUDA

We further compare CPU and CUDA under optimized algorithm. As shown in Fig 7, for small size problems (road length < 10 km), CPU is actually faster than CUDA. This is majorly due to the overhead of memory copy required in CUDA. When the size of problem gets larger, CUDA starts to dominant because memory copy is no longer a major part of execution time. For the 100km case, CUDA is roughly 8x faster than CPU.

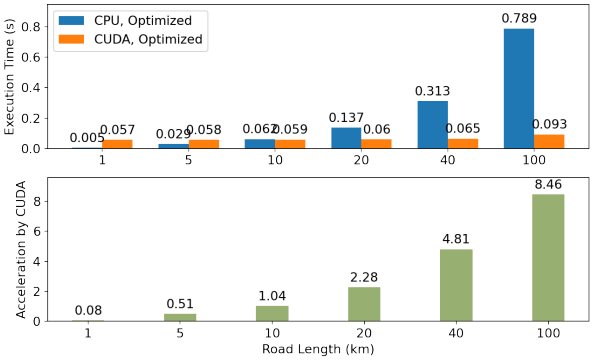

Fig. 7: Top: Comparison of execution time between CPU and CUDA. Bottom: Acceleration by CUDA

D. Compare CPU and CUDA with Lane Changing

For simplicity, the above results were all compared without activating the lane changing functions/kernels. Since lane changing requires more memory loads and operations, we expect CUDA gets more dominant with lane changing. From Fig 8, we can see that the maximum acceleration increases from 8.46 (Fig 7) to 10.2 (Fig 8). However, such an increase is not considered to be much. This signals some bottleneck in our optimized algorithm, which will be addressed in the next section.

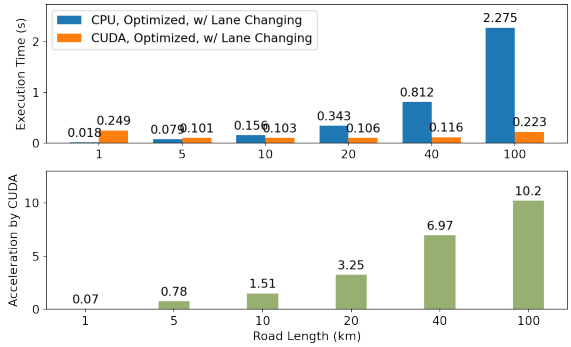

Fig. 8: Top: Comparison of execution time between CPU and CUDA with lane changing. Bottom: Acceleration by CUDA with lane changing

VI. Conclusion

In this article, we implemented both CPU serial and CUDA (GPU) parallel algorithms for agent-based traffic simulation. We compared a naive algorithm and an optimized algorithm. The results showed that the optimized algorithm is generally 2-3x faster than the naive one. While both CPU and CUDA algorithm get faster after being optimized, the reasons are believed to be different: CPU is generally benefit by reduced *For* loops, CUDA is benefit from less warp-divergence.

As a key part of this article, we compared how much CUDA is faster than CPU under the optimized algorithm. We find CPU is faster under problems with small size, but CUDA gets much more faster when the size increases. In our setup, the maximum acceleration by CUDA is found to be around 10x.

When we include lane changing in our model, we expect better performance for CUDA since more memory loads and operations are required. However, the results showed similar performance with and without lane changing. This may imply a bottleneck in our algorithm and triggers the necessity of future works. A guess of the bottleneck is that, for both CPU and CUDA, we need to check the cells one-by-one to determine safety condition for lane changing. This slows down the entire program. Three preliminary methods are proposed to solve this issue:

- Use a more comprehensive mapping and sorting algorithm to avoid one-by-one cell checks.
- Use a concurrent parallel kernel to check the cells. This is "parallel inside parallel".
- Use shared memory to store cell information if the reuse is intense, e.g. if the traffic density is high.

Overall, we present an efficient parallel algorithm for agent-based traffic simulation with around 10x acceleration. We expect further improvements in the future.

APPENDIX I
TIME STEPS FOR CONVERGING TO EQUILIBRIUM

In Table III we fixed the simulation steps to be 100 because we find that the system typically converges to an equilibrium state within 100 time steps. Fig 9 shows an example of how the average speed v_m converges over time. In this case, v_m reaches a plateau after around 40 time steps.

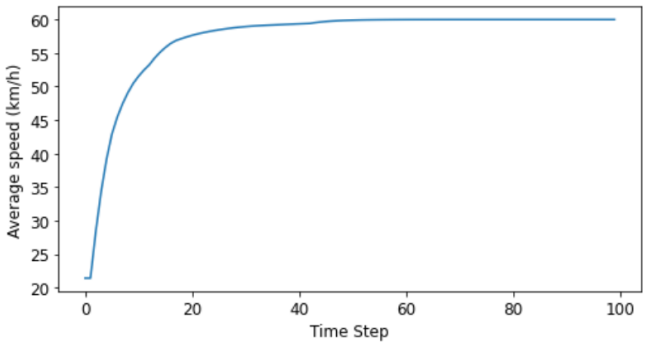

Fig. 9: A typical curve of average speed v_m

APPENDIX II
GITHUB REPOSITORY

The implementation along with the instructions and demo can be found here: https://github.com/zuzhaoye/traffic-sim-cuda

REFERENCES

[1] Nagel, Kai, and Michael Schreckenberg. "A cellular automaton model for freeway traffic." Journal de physique I 2.12 (1992): 2221-2229.

[2] D'Souza, Roshan, et al. "Data-parallel algorithms for agent-based model simulation of tuberculosis on graphics processing units." SpringSim. 2009.

[3] Hirabayashi, Manato, et al. "Toward GPU-accelerated traffic simulation and its real-time challenge." (2012).

[4] Shen, Zhen, Kai Wang, and Fenghua Zhu. "Agent-based traffic simulation and traffic signal timing optimization with GPU." 2011 14th International IEEE Conference on Intelligent Transportation Systems (ITSC). IEEE, 2011.

[5] Saprykin, Aleksandr, Ndaona Chokani, and Reza S. Abhari. "GEM-Sim: A GPU-accelerated multi-modal mobility simulator for large-scale scenarios." Simulation Modelling Practice and Theory 94 (2019): 199-214.

[6] Xiao, Jiajian, et al. "Pedal to the Bare Metal: Road Traffic Simulation on FPGAs Using High-Level Synthesis." Proceedings of the 2020 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation. 2020.

[7] Kosiachenko, Lisa, Nathaniel Hart, and Munehiro Fukuda. "MASS CUDA: a general GPU parallelization framework for agent-based models." International Conference on Practical Applications of Agents and Multi-Agent Systems. Springer, Cham, 2019.