# Accelerating Traffic Simulation and Its Real-Time Challenges

Manato Hirabayashi, Shinpei Kato, Masato Edahiro
*Department of Information Engineering*
*Nagoya University*

Yuki Sugiyama
*Department of Physics*
*Nagoya University*

## Abstract

*PLACEHOLDER.*

## 1 Introduction

Transportation systems underlie our industry and life as part of societal infrastructure. Economy depends highly on traffic flow. Once a traffic jam occurs, for example, it could cause significant economic damage through degradation of transport efficiency, energy consumption, and environmental poisoning. According to the Japanese government report a few years ago, economic losses due to traffic congestion reach one hundred billion dollars per year in Japan. Albeit a major issue of transportation systems, the mechanism of traffic jam is not well-explained in the literature. What is particularly important is "phantom jam", which often happens on freeways without any accident. This phantom jam should be avoided by science, given that it is a physical phenomenon but not one caused by human errors. However, we are not aware of scientific and engineering solutions to such real-world problems.

In physics, traffic flow is described by mathematical models [1, 3, 5]. While their mathematical expressions are not identical, they are all compute-intensive procedures. For example, traffic simulation based on the Optimal Velocity (OV) model [1] computes locations and velocities of agents every sampling period, solving OV equations. Specifically, the location $x_n$ of the $n$th agent is described by the following equation, where $\Delta x_n = x_{n+1} - x_n$ is a distance to a preceding agent, $a$ is a sensitivity, and $V()$ is an optimal velocity function:

$$\frac{d^2 x_n}{dt^2} = a \left\{ V(\Delta x_n) - \frac{dx_n}{dt} \right\}. \tag{1}$$

Applying a large number of agents to simulation using the above formula, it is apparent that computational workload increases exponentially. In fact, the above formula considers only one dimension, which restricts applications of simulation to freeway traffic flows, powder flows, molecular motors, and so on. Making it multi-dimensional further increases computational workload, while allowing more complicated simulations, such as traffic networks, internet packet flows, evacuation routes, and herd formations of animals, to use similar optimal velocity models. Given a scale of million agents in the real world, traffic simulation should be supported by powerful computer systems.

Deployment of traffic simulation may require real-time feedback from the real world. This turns out to be a cyber-physical systems (CPS) problem. Real-time traffic simulation is another challenging issue, where the rate and the preciseness of simulation must be traded to meet the requirement of a given scenario. For instance, evacuation may want very high rate simulation, sacrificing the preciseness of simulation to some extent. Unfortunately, none of those challenges has been explored yet, largely due to a lack of multidisciplinary research collaborations between physics and computer science.

This paper explores how to accelerate traffic simulation using the state-of-the-art parallel computing technology. In particular, we use the graphics processing unit (GPU), which integrates hundreds of cores on a chip. Recent GPUs are becoming more and more suitable for general-purpose data-parallel applications. The traffic simulation program used in this paper applies Equation (1) to a large number of agents. The resulting workload is highly data-parallel and compute-intensive, which can be nicely offloaded on to the GPU. We also identify the bottleneck in accelerating our traffic simulation program using the GPU, and provide some insight into its solution.

The rest of this paper is organized as follows. Section 2 describes the assumption and terminology used this paper. Section 3 briefly explains our traffic simulation program. Section 4 presents our schemes of GPU implementations of traffic simulation. Section 5 evaluates the benefit of our GPU implementations, and provides an insight into future work. Section 6 concludes this paper.

## 2 Assumption and Terminology

We assume the Compute Unified Device Architecture (CUDA) for GPU programming. In CUDA, a unit of pieces of code that is launched on the GPU is called a *kernel*.

The kernel is typically composed of multiple parallelized *threads* that accelerate computations. A unit of threads is called a *block* where most computing resources are shared. A collection of blocks for the corresponding kernel is called a *grid*. The maximum number of threads that an individual block can contain is defined by the GPU architecture.

GPU applications use a set of the API supported by CUDA. We typically take the following steps to use the GPU: (i) allocate space to device memory, (ii) copy data to the allocated device memory space, (iii) launch the program on the GPU, (iv) copy resultant data back to host memory, and (v) free the allocated device memory space.

## 3 Traffic Simulation

PLACEHOLDER.

1. Initialize the time $t$, and set the initial values of $x_n(t)$ (also $y_n(t)$ and $z_n(t)$, if necessary for multidimensional versions).

2. Increase the time $t$ by the sampling period $\Delta t$.

3. Compute the location $x_n(t)$ (also $y_n(t)$ and $z_n(t)$, if necessary for multidimensional versions), and the velocity $v_x(t)$ at time $t$ for each agent $A_n$, using the OV model.

4. Go back to Step 2, if the simulation time is expired.

5. Exit the program.

Note that Step 3 exploits a lot of loop procedures to derive the locations and velocities of agents. This part can be accelerated by the GPU.

## 4 GPU Implementation

GPU performance is dominated by the program design. GPU-accelerated programs are typically divided into two pieces of code. The CPU code plays a role of a master thread that controls the program flow. The GPU code, on the other hand, spawns a bunch of worker threads to execute compute-intensive parts of the program in parallel, thus accelerating the overall program. What is often argued in performance optimization is how to parallelize the compute-intensive parts into threads. This is actually the well-studied problem in the literature. What is not really understood yet is when to offload the program on to the GPU. This paper explores two schemes that use the GPU at different timings to see how GPU performance is affected.

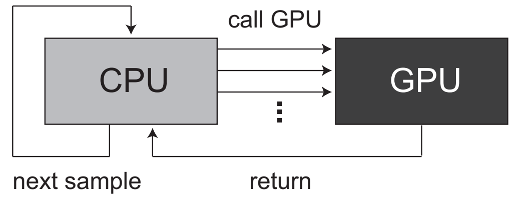


**Figure 1. Block diagram of the Sample-in-CPU scheme.**

### 4.1 Sample-in-CPU Scheme

We first implement such a scheme that uses the GPU *only if necessary*. In other words, the program is offloaded on to the GPU, only when computations of locations and velocities of agents are accelerated by parallelization. The CPU bridges across sampling periods to manage simulation. In this scheme, the control flow of simulation is always returned to the CPU at the end of each sampling period. Such a synchronized approach makes the programmer easy to obtain intermediate results of simulation, whereas the overhead imposed on moving back and forth between the CPU and the GPU must be compromised.

Figure 1 shows a brief overview of the Sample-in-CPU scheme. This scheme also has several alternatives depending on how many kernels need to be launched on the GPU in each period. Suppose that we want to offload $n$ pieces of *for* loops on to the GPU. We may return to the CPU $n$ times in totall, that is, return every time one *for* loop breaks, or otherwise we may just return once when all the $n$ *for* loops end. This is a design decision, and is also dependent on the GPU architecture and the parallelization structure. In general, parallelized threads need to be synchronized when moving across basic blocks. Specifically, when moving to the next *for* loop executed in parallel, all the threads relevant to this parallel computing procedure must synchronize with each other. However, the maximum number of threads that can be synchronized on the GPU is often limited. As of 2012, for example, NVIDIA's GPU architectures limit the number of such threads to 1024 or less [8]. Our implementation therefore forces the program to return to the CPU every time one *for* loop breaks. Note that this is not a conceptual limitation of GPU computing, but is a current limitation of hardware. We believe that this limitation would be removed or mitigated in next-generation GPU architectures.

### 4.2 Sample-in-GPU Scheme

We next implement such a scheme that uses the GPU *all the time*, even to control simulation. There is one big kernel running on the GPU, which is launched only once at the beginning. After offloading the program on to the GPU, the

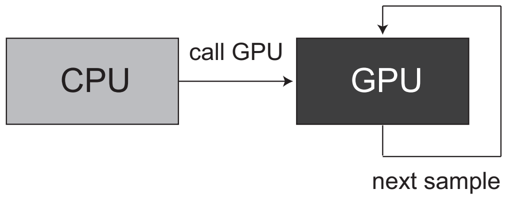


**Figure 2. Block diagram of the Sample-in-GPU scheme.**

CPU is going to wait for the completion of simulation. This scheme is almost optimized in performance, since there is little overhead in communication between the CPU and the GPU. The downside of this scheme is that the CPU and the GPU are not synchronized. The progress of simulation is not visible from the CPU, unless the program implements a specific interface to allow the CPU to access intermediate results of simulation running on the GPU. In our implementation, we consider providing a framework that the CPU downloads data from the GPU asynchronously without awareness of simulation, when the user requests intermediate results. This asynchronous data access may affect the performance of simulation, though in reality this access would not happen more than once in a sampling period.

Figure 2 shows a block diagram of the Sample-in-GPU scheme. The procedure is very simple. The most portion of code of simulation is executed on the GPU. Given that the single-thread performance of recent GPUs is getting more reliable, this approach is pretty reasonable. As mentioned in Section 4.1, however, the current limitation of the GPU architecture prevents us from synchronizing among blocks at a scale of thousands threads. Therefore, this scheme is speculative in a sense that it does not work today but may appear in the future.

This paper provides some degree of insights into how this scheme is effective. We implement this scheme with the current GPU architecture under the assumption that global synchronization among blocks works. There is also another possible approach for this scheme that we limit the number of threads to what is supported by the GPU architecture. This alternative implementation, however, is left open for future work.

## 5 Evaluation

We evaluate the performance of our GPU-accelerated traffic simulation programs, using an NVIDIA GeForce GTX 560 Ti graphics card. This is a middle-end graphics card based on the Fermi architecture [6], integrating 394 compute cores on a chip. The programs used in this evaluation are written in CUDA, and are compiled by the NVIDIA CUDA Compiler (NVCC) v4.2 suite [7].

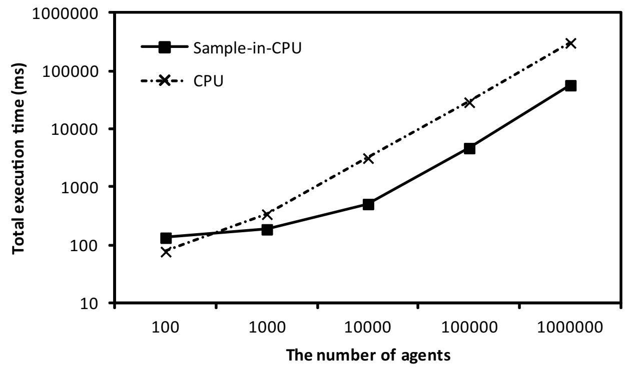


**Figure 3. Performance benefit of the GPU.**

We focus on the one-dimensional OV model [1], which can be applied to freeways simulation. The original program [9] of this simulation is written in C, without any parallelization. We use the same simulation parameters as the original default setup, while increasing the number of agents (cars) to see how simulation performance varies. The evaluation is conducted by comparing our GPU implementations with the original CPU implementation.

### 5.1 Performance Benefit

Figure 3 demonstrates the performance benefit of the GPU under the current hardware limitation. The Sample-in-CPU scheme herein exploits synchronization between the CPU and the GPU after every *for* loop to overcome the limitation that CUDA threads cannot synchronize beyond blocks. Despite of this restricted implementation, our GPU implementation outperforms the original CPU implementation by about five times in simulation time, when the number of agents exceeds a scale of 1000. This evinces an affinity of the GPU and parallel computing for traffic simulation. It is also interesting to see that the CPU implementation has a better performance for a small number of agents. Thus, the overhead of communication between the CPU and the GPU has non-trivial impact if the achievable parallelism is not sufficient. Another notable observation is that the diffence in performance of the CPU and the GPU implementations is saturated when the number of agents reaches 1000. This saturation implies the hardware potential of the NVIDIA GeForce GTX 560 Ti graphics card used in this evaluation. Using high-end graphics cards, the difference in performance may not be saturated at the same scale.

### 5.2 Impact of Hardware Limitation

Figure 4 demonstrates the impact of the current hardware limitation that CUDA threads cannot synchronize beyond

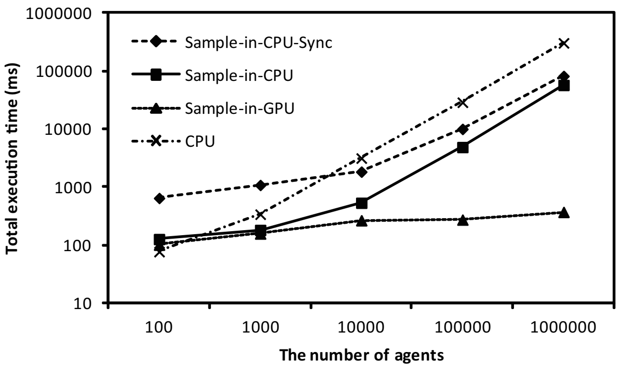


**Figure 4. Impact of the hardware limitation.**

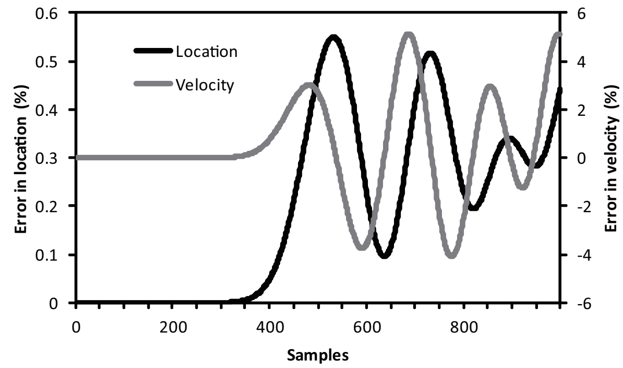


**Figure 5. Error in location and velocity due to imprecise synchronization.**

blocks. In this experiment, we observe "what would happen if this hardware limitation is removed?", compromising the preciseness of simulation results. Namely, we use the current CUDA synchronization function to synchronize threads running in different blocks, even though it does not work as expected. All the GPU implementations of the plotted schemes use this assumption. Therefore, the Sample-in-CPU scheme is now different from what we have evaluated in Figure 3, since there is no need anymore to return to the CPU every time one *for* loop breaks. We instead put a synchronization function between *for* loops on the GPU. The Sample-in-CPU-Sync scheme is an alternative version of the Sample-in-CPU scheme in that we download intermediate results from the GPU to the CPU at the end of each sampling period. The Sample-in-GPU scheme is what is presented in Section 4.2.

It is very notable that the Sample-in-GPU scheme improves the simulation time by a factor of 1000 as compared to the original CPU implementation. This result encourage suport for thread synchronization among blocks in CUDA programming. There are also several interesting observations obtained in this experiment. Comparing the Sample-in-CPU and the Sample-in-CPU-Sync schemes, it turns out that the cost of downloading intermediate results from the GPU to the CPU is becoming trivial as the scale of simulation increases. Hence, a common argument of "host-device data communication in GPU programming is expensive" does not apply to large-scale traffic simulation. Another issue of concern often discussed in GPU programming is that a single-thread performance of GPUs is weak as compared to that of CPUs. This is true in general, but our experimental result shows that the performance loss caused by pushing most pieces of code into the GPU in the Sample-in-GPU scheme is not signficant even for a small number of agents. This observation leads to some conclusion that the Sample-in-GPU scheme would be the best choice for any scale of traffic simulation.

### 5.3 Practical Concern

We finally examine the impact of misbehavior of thread synchronization in our GPU implementations. The simulation results produced by our GPU implementations are not precise except for those of the Sample-in-CPU scheme presented in Figure 3, due to our optimistic use of thread synchronization on the GPU. We need to wait for new GPU architectures removing the limitation of thread synchronization in order to make them precise. Practically speaking, however, imprecise results of simulation are still valuable and meaningful as far as they are limited to an acceptable range of errors for traffic flow.

Figure 5 demonstrates the error percentage of location and velocity between the simulation results produced by the original CPU and the imprecise Sample-in-GPU implementations. Since this is one-dimensional OV simulation, there are only forward or backward in location, and high or low in velocity. Under this constraint, the error in location is very trivial. This is because the characteristic of traffic flow is well captured, even though the simulated locations and velocities have some error in their individual values. A degree of the error in velocity is slightly high as compared to that in location. This means that velocity is more sensitive to imprecise results than location. However, the maximum error in velocity observed in our experiment is at most 5%. Suppose that the average driving speed is 50mph. The simulated speed in our imprecise implementation would still be within a range of 47.5 to 52.5mph. Intuitively, this scale of error is acceptable in practice. We plan to further investigate if the same argument can be applied to a more variety of parameter setups and multidimensional OV simulation programs.

## 5.4 Discussion

PLACEHOLDER.

Imprecise Computation [4]. Gdev [2].

# 6 Conclusion

PLACEHOLDER.

# References


[1] M. Bando, K. Hasebe, K. Nakanishi, A. Nakayama, A. Shibata, and Y. Sugiyama. Phenomenological Study of Dynamic Model of Traffic Flow. *Journal de Physique I France*, 5(11):1389–1399, 1995.

[2] S. Kato, M. McThrow, C. Maltzahn, and S. Brandt. Gdev: First-Class GPU Resource Management in the Operating System. In *Proc. of the USENIX Annual Technical Conference*, 2012.

[3] B.S. Kelner and P. Konhauser. Cluster Effect in Initally Homogeneous Traffic Flow. *Physical Review E*, 48:2335–2338, 1993.

[4] K. Lin, S. Natarajan, and J.-S. Liu. Imprecise Results: Utilizing Partial Computations in Real-Time Systems. In *Proc. of the IEEE Real-Time Systems Symposium*, pages 210–217, 1987.

[5] K. Nagel and M. Schrekenberg. A Cellular Automaton Model for Freeway Traffic. *Journal de Physique I France*, 2(12):2221–2229, 1992.

[6] NVIDIA. NVIDIA's next generation CUDA computer architecture: Fermi. `http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architechure_Whitepaper.pdf`, 2009.

[7] NVIDIA. CUDA 4.2. `http://developer.nvidia.com/cuda/cuda-downloads`, 2012.

[8] NVIDIA. NVIDIA GeForce GTX 680: The fastest, most efficient GPU ever built. `http://www.geforce.com/Active/en_US/en_US/pdf/GeForce-GTX-680-Whitepaper-FINAL.pdf`, 2012.

[9] The Mathematical Society of Traffic Flow. Optimal Velocity Model. `http://traffic.phys.cs.is.nagoya-u.ac.jp/~mstf/sample/ov_e.html`, 1995.