GPU optimized parallel implementation of NaSch traffic model

Yassine El Hafid, Abdessamad El Rharras, Mohamed Wahbi and Rachid Saadane
{yassine.el.hafid,a.elrharras,wahbi.mo,rachid.saadane}@gmail.com
SIRC/LaGeS EHTP, EHTP
Casablanca, Morocco

Abstract. Traffic simulation has a practical interest for modern society, mainly in minimizing the problems of congestion, which is the main cause of road accidents. In this way, several traffic models were developed based on Cellular Automata (CA). Moreover, the simulation of traffic system requires a high computing ability. Therefore, this paper explores the ways to use the graphics-processing unit (GPU) for reducing time simulation of such applications. We propose several software implementations to maximize GPU performance for NaSch model parallelization. Compared to CPU, the simulation results show that we need optimization effort to get better performance results for GPU.

Keywords: GPGPU, CPU, CUDA, NaSch model, Road Traffic, Parallel computing optimization

1 Introduction

Road traffic is considered as an important economic lever which has a significant impact on modern society. Much research is carried out in order to reduce the problems of congestion, road accidents, or the creation of appropriate infrastructure in a given socio-economic context. Most of the information collected on traffic is based on empirical measurements by proximity sensors, traffic video or aerial and satellite images. The measurements are always done on a set of disparate vehicles in uncontrolled environments. Thus, with the development of information technology tools, the interest of scientists for numerical simulations is quickly established. Several traffic models were developed to approximate the actual drivers behavior and the conditions in which they operate. This type of simulation requires significant computing capability, causing a significant latency time ranging from several hours to several days. Additionally, new processors architectures were immersed in recent years. Designed initially for video games, graphic processors know considerable improvements in performance calculations, number of cores and memory bandwidth integrated in GPU cards. Indeed, some newer graphics cards have up to 5760 calculations cores and up to 24 GB of dedicated RAM; with a relatively affordable price for scientists. Several generations of processors were releases with significant improvements in processor architecture, power consumption and the software framework, allowing access to the features of the GPU. In this context, we took an interest in GPU to implement a basic traffic model developed by

ICCWCS 2019, April 24-25, Kenitra, Morocco

DOI 10.4108/eai.24-4-2019.2284233

K. Identify applicable funding agency here. If none, delete this. Nagel and M. Schreckenberg [1]. The GPU could handle a very big number of interacting cars that require memory space and high-speed calculations. Therefore, algorithm optimization to the target GPU is a first step in simulating a large number of cars drivers behaviors. Then, in the second section of this paper, we present different models and methods for road traffic especially NaSch model and we introduce the architecture of Nvidia Cuda parallel programming. The third one, we present detailed implementations of NaSch model, sequential on CPU and parallel on GPU. In the fourth part, we show the experimental results and a discussion of their interpretation. Finally, we conclude the paper and we propose future prospects of our work.

2 Introduction Methods and models for road trafic

2.1 Trafic models

To describe, understand and improve road traffic, studies provide mathematical models inspired from fluid mechanics, which describe the different interactions between the driver and his vehicle. These models include the mobile elements (vehicles, pedestrians ...) into a representation of the entire road system (roads, traffic, steering wheel ...). These studies are helpful to investment in infrastructure. The first traffic studies are born before the Second World War, they are based on an Adams approach [2] using the theory of probability, and on the work of Bruce D. Greenshields (Yale Office of Road traffic) [3] for modeling the evolution of flows, speeds and traffic at intersections. After the war, researchers were interested in tracking the vehicle [4] [5], they developed a theory of traffic flow [6] [7] and the Queueing theory [8]. Later, in the early 1970s, Payne and Whitham [9] were pioneers in proposing a new approach based on the fluid flows from fluids mechanics. More recently, while the car park grows, scientists try to improve the existing methods and propose new models more consistent with the current traffic conditions. To highlight the

Fig. 1. Cyclical road to simulate an infinite highway.

different principles and attempt a classification among all the models, Hoogendoorn and Bovy [10] propose to take the level of detail in modeling (individual element) as classification criteria to determine four groups:

- Microscopic models describe each component of the system in their behaviors and interactions in a high level of detail, for example a vehicle and its driver [4];
- Models submicroscopic, further the level of detail cited above, disclose the operated controls (indicators, speed changes ...) within the vehicle and put them in relation with the environmental conditions [10] [11];
- The mesoscopic models represent traffic in a lower level of detail by small ensembles whose behaviors are described in terms of probabilities [12] [13];
- Macroscopic models treat the traffic in a more global perspective and describe traffic in terms

of speed, flow and density; the specific characteristics of the fluid flow [6] [13].

- **2.2 NaSch model**

NaSch model [1] is a cellular automata model [14] where a section of the road is divided into cells. Each cell has a specific length of 7.5 meters, which expresses the sum of the length of most types of vehicles with inter-vehicle distances, spacing a vehicle and its two nearest neighbors in the case of a blocked traffic. This value of the minimum space occupied by a vehicle is obtained by the inverse of the blocking density 133 veh/Km, measured empirically. A cell is either empty or occupied by a single vehicle. The possibility of accidents is null and overtaking is not allowed. An iteration of NaSch corresponds to a movement of vehicles in a one-second time interval, by a four-phase algorithm. The minimum speed is 7.5 m / s (27 km / h) corresponding to a speed V =1. Moreover, other speed values are integer multiples of 7.5 m/s. As for acceleration, it takes a maximum value of 7.5 m / s (0.76 g). For the simulation, we consider a road that contains L cells, a closed circuit as illustrated in Figure 1.

- PHASE I: ACCELERATION In this phase the vehicle "i" is accelerated by a unit without exceeding the prescribed maximum speed. As well, this phase is given by equation (1).

$$V_i'(t) = \min(V_i(t) + 1, V_{max}) \quad (1)$$

- PHASE II: DECELERATION Since the vehicle "i" is not alone on the road, we must take into account the vehicle $i + 1$ in front of him, whose position is $X + 1(t)$. As well the movement of the vehicle i does not exceed the vehicle position $i + 1$ according to Equation (2).

$$V_i''(t) = \min\left(V_i'(t), d(t)\right) \ \ with\ d_i(t) = x_{i+1}(t) - x_i(t) - 1 \quad (2)$$

$$V_i''\ (t) = min(V_i'\ (t), d(t)) \text{ with } d_i(t) = x_{i_{+1}}(t) - x_i(t) - 1$$

- PHASE III: RANDOMIZATION Equation (3) allows simulating the behavior of drivers in an approximated way, and the speed is decelerated according to the probability *p*.

$$\text{If } V_i''\ (t) > 0 \quad \text{then} \quad (3)$$

$$\begin{cases} V_i(t+1) = V_i''(t) - 1 \text{ with a probability } p \\ V_i(t+1) = V_i''(t) \text{ with a probability } 1 - p \end{cases} \quad (4)$$

- PHASE IV: POSITIONS UPDATING Each vehicle is moved forward, taking into account the speed previously calculated.

$$x_i(t + 1) = x_i(t) + V_i(t + 1) \quad (5)$$

Figure 2 summarizes the four steps in Nash and Figure 3 shows the results that we

obtained by numerical simulation:

2.3 NVIDIA Parallel computing solution

1) The Architecture of NVIDIA GPU:: The first GPU card was released in 2007 called TESLA. Several generations have followed with increasing performances in: memory size, cores speed, the bandwidth of memory and finally, the pipelines configuration of tasks or threads execution. Nvidia made a classification of these processors according to their computing capabilities. There are major differences from one generation to another and even in different versions

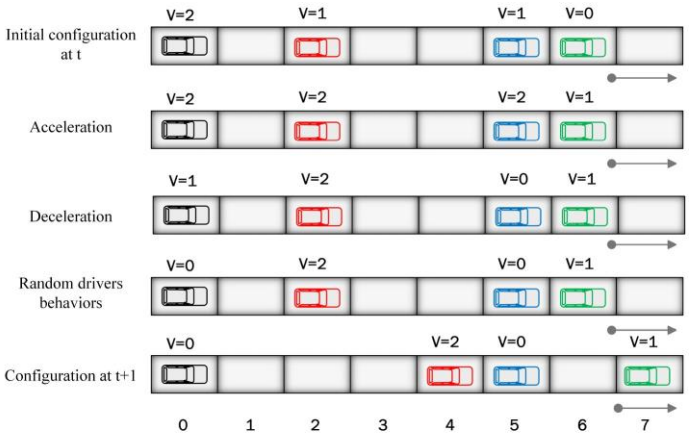

Fig. 2. Moving vehicles in the four NaSch phases.

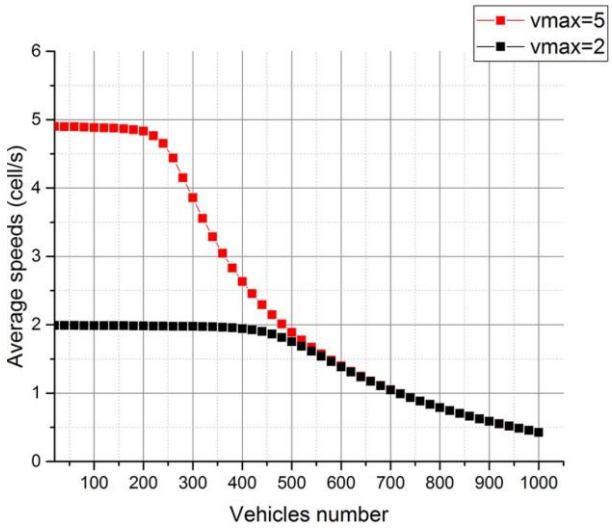

Fig. 3. Average speeds based on the number of vehicles.

of the same generation architecture. Some specific features of the processors categories gives

developers alternatives of parallelization that could reduce significantly the computation time for certain algorithms. A basic solution of GPGPU consists of a main processor CPU named Host, and an auxiliary processor GPU named Device. Each processor has its own RAM. Thus, the host has a main DDR3 RAM, and the Device has GDDR5 RAM. Communication between the two processors is done by PCIE bus. The figure 4 shows the simplified architecture of a GPU on a graphics card. The various types of memory available to the processor enable fast execution of instructions. The main computer language used is C, but it is possible to use other languages like C++, FORTRAN, Java and Phyton. However to access the functionality of the GPU we use an associated API called CUDA (Compute Unified Device Architecture). The manufacturer NVIDIA supplies this parallel computing platform. Thus, in comparison with third-party solutions, CUDA has the advantage of being improved continuously and especially to be optimized for different generations of GPUs. In a Cuda program, there are two types of code: the first, which is sequentially executed by the host, this code call device services via functions named kernels. The Device cores execute threads that are the images of these functions Kernels (second type of code) in parallel and asynchronous way. At the level of software, the threads are banded together in blocks, which their turns are grouped into a computing grid. Each grid corresponds to a kernel called by the host. The number of threads and blocks depends on the programmer choices and hardware limitations on used GPU generation. Threads can access multiple types of memory available to the GPU as shown on the figure 5.

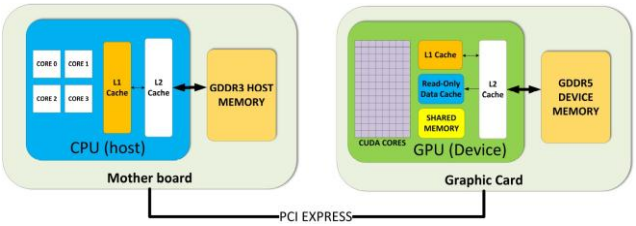

Fig. 4. Hardware architecture of GPU-based high performance computing (HPC).

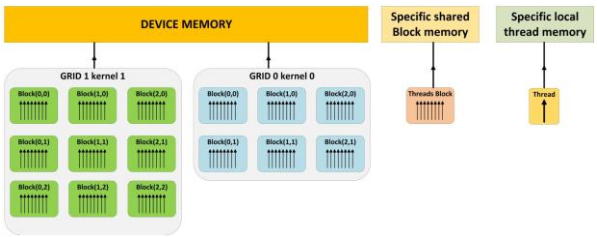

Fig. 5. Programming model organization of GPU parallel computing.

Hardware Implementation: The streaming multiprocessor (SM) is a group of computing cores; their number depends on the generation and the version of the GPU chip. It creates, manages, plans, and executes in parallel several groups of 32 threads called warp. The figure 6 illustrates an SM of Kepler architecture, which has four warps. The threads of the same warp begin together at the same program address. They have their own state registers, and their own counter instruction address, which gives them the ability to run independently. When we give to multiprocessor one or more blocks of threads to be executed, it distributes them into warps then they are organized by warps schedulers for their executions. The way a block is divided into warps is always the same; each warp contains consecutive threads, of threads whose identifier is incremented from the first warp containing the thread 0. A warp executes one common instruction at the same time; the maximum performance is achieved when all 32 threads in a warp follow the same execution path. If the execution of a warp threads diverge via a data-dependent conditional branching, then the warp runs sequentially each path branching, while the threads that do not follow the path are disabled. When all the paths are complete, the threads converge back to the same execution path. The divergence of threads execution occurs only in a warp; different warps run independently, regardless that they execute commons or severed code sequences. The threads in a warp sequentially execute certain type of instruction as atomic operations.

2 Implementation

3.1 Data model

A. Parallel Implementation

Like the sequential implementation, we adopted the same program and data structure. The major difference lies in the calculation delegation of four NaSch phases to the GPU. Moreover, Figure 8 summarizes the first implementation giving unexpected performance results. This fact led us to improve our implementation in stages. Among the experienced improvements, we use atomic operations and reduction of the requests number from CPU to the GPU (CUDA launch). It thus reached a program structure better optimized as shown in Figure 9. In addition, we test several configurations of calculations distributions cores to determine the optimization rules that will be used in future implementations. For each implementation three kernel functions are used, two of which are common to both GPU implementations: setupkernel and load configcar. A third function has two versions, and traficsimulationver2 trafic simulationver3, each applied respectively. A fourth, CalculVmoy, calculates the average speed of the vehicles and used only in the first implementation. The equivalent of this function has been completely merged in the second implementation, which in fact represent the second detailed optimization in the discussion section. The function TraficsimulationVer2 is loaded excessively by the host through the Calculpoint function. However, in traficsimulationver3 we implement Monte Carlo loop, NaSch iterations and speed average calculations within the kernel function, in this way we reduce the time cost of CUDA functions calls.

Fig. 6. Kepler SM architecture [15].

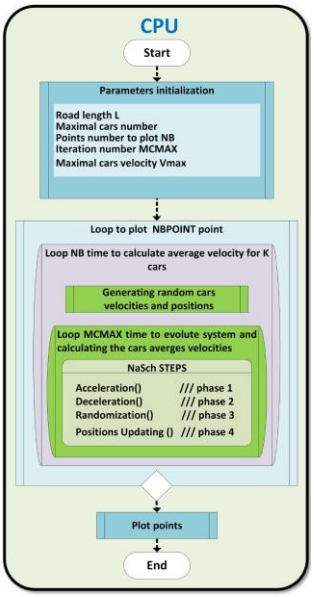

Fig. 7. Sequential implementation on CPU.

B. Sequential implementation

The sequential implementation illustrated in Figure 7 begins with the determination of the simulation parameters, in which we specify the number of cells, the maximum number of vehicles, the number of plotting points, etc. Then three nested loops execute the NaSch iterations to plot the traffic characteristic curves. Moreover, at each Monte Carlo iteration, we reinitialize position and speed parameters for the vehicles. In addition to resetting the configuration, we introduce captor table for measuring the flow in a given position depending on the number of vehicles.

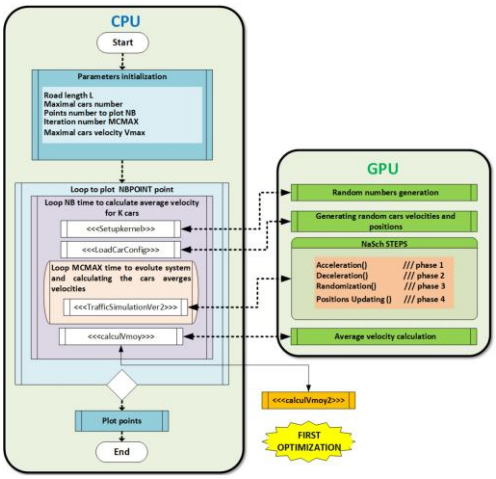

Fig. 8. The first implementation and first optimization for GPU.

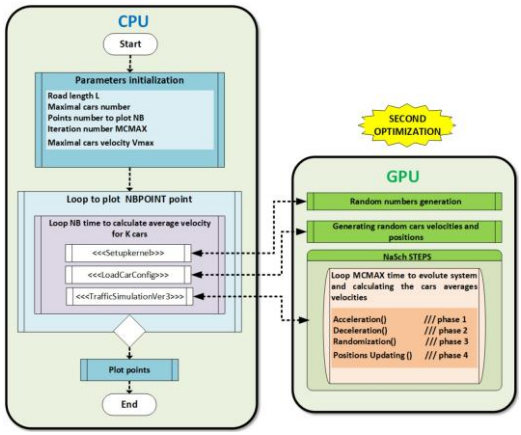

Fig. 9. The second version of implementation for GPU.

3 Result and discussion

3.1 First GPU optimization

As announced in the previous section, the results are the opposite of what we planned. Clearly seen in Figure 10, the CPU is faster compared to the GPU. Therefore, we tried to improve performance by using atomic operations. Indeed, at the beginning a kernel function called CalculVmoy() was used which calculates sequentially the average speed vehicles. The problem with this implementation is that before calculating the average, we have to wait execution of the warp containing the concerned thread. The second version of the function CalculVmoy2() uses the atomic function atomicAdd () that allows different threads from different warp to add the intrinsic speed car. Although this operation is done sequentially on the warp, deferred executions of warps based on the principle of first come, first served, reduces the relatively long waiting time of the previous method.

Moreover, we used a shared variable type to perform summation, giving more efficiency for atomic operations compared to the case of using global memory type variables; this is due to the big latency time to access to this memory type.

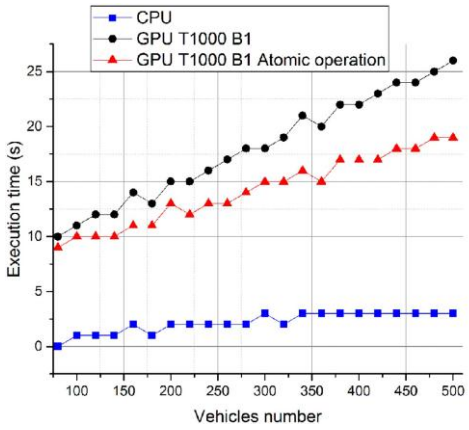

Fig. 10. Execution time CPU versus GPU Implementation 1, compared with GPU Implementation 1 using atomic operation.

A. Second GPU optimization

Despite the improvements obtained by the first implementation, the execution time remains long compared to the CPU. Thus, by using the performance analysis tool Nsight, it was noted that significant time is consumed by the Runtime API functions, specifically the CudaLaunch function to launch kernels. This is why we elaborate a second implementation.

By bringing Monte Carlo loop within the kernel function traficSimulationVer3, temporal cost of kernels launches was reduced, as shown in Figure 11. Again, the performance is not up to what is expected. We noted a significant performance gain for a relatively small number of vehicles. Nevertheless, the two curves are joined to values approximating thousand vehicles. After analyzing our implementation, we note that an atomic function is requested excessively. This function calculates the average vehicle speeds in a single iteration. To address this problem, we adopt the following idea: each vehicle (thread) accumulates its speed throughout the iterations and in the end; we make atomic summation of all values. Figure 12 reflects the improvements achieved in the second optimization. We remark the clear advantage of the GPU compared to CPU and the previous GPU implementations. Moreover, we note that the CPU and GPU curves intersect, as in the previous section. An important point to remember is that we use 1000 threads per block for time measurements in both implementations. Thereafter we explore more computing distribution configurations given the results shown in Figure 13. Thus, for 2000 vehicles, there are significant performance losses in the configuration of two threads by 1000 blocks. Although, the use of a large number of blocks means using more processing cores, so we observe a significant reduction in performance by increasing vehicles number. This could be explained by using the second atomic function, which computes the average speed based on those previously calculated by threads block. Because a block cannot access to the shared memory of its neighbors, the atomic operation is performed with global variable. The time required is therefore more gradually as the number of blocks is increased.

4 Conclusion

The general problem with GPU programming is the need to have strong technical skills to success a parallel implementation of a given algorithm. Thus, in this work we have implemented an evolving version of the NaSch algorithm. The constraints imposed by the traffic model led us to propose improvements in several stages to achieve results in line with our expectations. The implementation methods have a significant impact on computing performance. Using atomic operations and reducing kernel launching has benefit to minimize calculation latency in GPU cores. As a future work we will use our developed model the investigate applications in Deep Learning and Machine Learning [16]. An other perspectives in relation to the algorithms used are cloud and edge computing

References

[1] Nagel, K. and Schreckenberg, M.: A cellular automaton model for freeway traffic. Journal de physique I, 2(12):2221-2229, 1992. [2] William F Adams. Road traffic considered as a random series.(includes plates). Journal of the ICE, 4(1):121-130, 1936.
[3] Greenshields, BD. Channing, Ws. and Miller, Hh.: A study of traffic capacity. In Highway research board proceedings, volume 1935. National Research Council (USA), Highway Research Board.
[4] A Pipes, L.: An operational analysis of traffic dynamics. Journal of applied physics, 24(3):274-281, 1953.
[5] Robert E Chandler, Robert Herman, and Elliott W Montroll.: Traffic dynamics: studies in car following. Operations research, 6(2):165- 184, 1958.

[6] J Lighthill, M. and Beresford, Whitham G.: On kinematic waves. A theory of traffic flow on long crowded roads. In Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences, volume 229, pages 317-345. The Royal Society, 1955.
[7] Paul I Richards. Shock waves on the highway. Operations research, 4(1):42-51, 1956. [8] L Saaty, T.: Elements of queueing theory, volume 423. McGraw- Hill New York, 1961.
[9] J Payne, H. Models of freeway traffic and control. Mathematical models of public systems, 1971.
[10] Hoogendoorn, S. P. and HL Bovy, P.: State-of-the-art of vehicular traffic flow modelling. Proceedings of the Institution of Mechanical Engineers, Part I: Journal of Systems and Control Engineering, 215(4):283-303, 2001.
[11] Brockfeld, E. Barlovic, R. Schadschneider, A. and Schreckenberg , M.: Optimizing traffic lights in a cellular automaton model for city traffic. Physical Review E, 64(5):056132, 2001.
[12] Maerivoet, S. and De Moor, B. Cellular automata models of road traffic. Physics Reports, 419(1):1-64, 2005.
[13] Leonard, DR. Gower, P. and Taylor. Contram, NB.: Structure of the model. Research report-Transport and Road Research Laboratory, (178), 1989.
[14] Makowiec, D.: The classification of homogeneous and symmetric cellular automata. Work, 1991, 1949. [15] NVIDIA. Nvidia kepler gk110 white paper, 2012.
[16] Abadi, M. et al. TensorFlow: A System for Large-Scale Machine Learning, 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16), November 2-4, 2016 • Savannah, GA, USA.

.

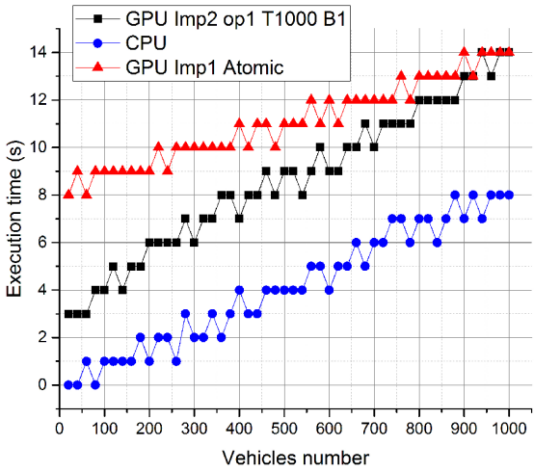

Fig. 11. Execution Time CPU versus GPU Implementation 1 atomic, compared to the second GPU optimization implementation 1.

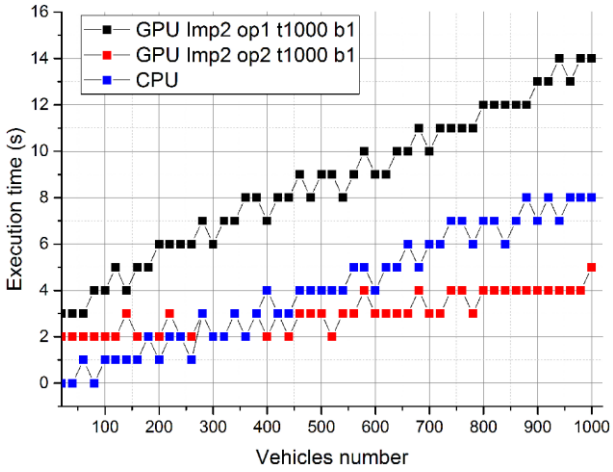

Fig. 12. Execution Time CPU versus GPU Implementation 2 optimization 1, compared to the second GPU optimization implementation.

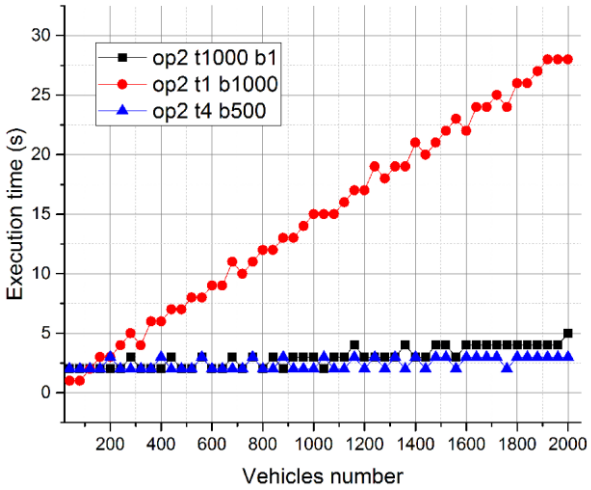

Fig. 13. Execution Time GPU Implementation 2 optimization2 with different configurations calculations distributions