Complexity

WILEY

RESEARCH ARTICLE OPEN ACCESS

Vehicular Traffic Networks Simulation: Towards the Implementation of a High-Performance Computing Simulator

Felipe Morales-Torres[1] | Eduardo G. Hernandez-Martinez[2] | Jose E. Quiroz-Ibarra[1] | Mauricio Flores-Geronimo[3] | Guillermo Fernandez-Anaya[4] | Enrique D. Ferreira-Vazquez[5] | Jose-Job Flores-Godoy[6]

[1]Institute of Applied Research and Technology, Universidad Iberoamericana Ciudad de Mexico, Mexico 01219, Mexico | [2]Division of Science, Art and Technology, Universidad Iberoamericana Ciudad de Mexico, Mexico 01219, Mexico | [3]Engineering Studies for Innovation Department, Universidad Iberoamericana Ciudad de Mexico, Mexico 01219, Mexico | [4]Physics and Math Department, Universidad Iberoamericana Ciudad de Mexico, Mexico 01219, Mexico | [5]Engineering Department, Universidad Catolica del Uruguay, Montevideo 11600, Uruguay | [6]Exact and Natural Sciences Department, Universidad Catolica del Uruguay, Montevideo 11600, Uruguay

Correspondence: Eduardo G. Hernandez-Martinez (eduardo.gamaliel@ibero.mx)

Received: 24 November 2024 | **Revised:** 16 December 2025 | **Accepted:** 2 February 2026

Academic Editor: Pramita Mishra

Keywords: GPU | high-performance computing | multiagent systems | object oriented programming | python | vehicular traffic network

ABSTRACT

This work presents a comparison of different software and hardware implementations of a traffic simulator using a fluid-based mesoscopic model. The model represents Vehicular Traffic Networks (VTNs) based on the flow resulting by the interconnection of dynamical agents associated to street segments and intersections. Starting from a single cross-intersection example, a traditional numerical simulation was developed in MATLAB as a first step. Afterwards looking for a leverage of the current computer capabilities, new approaches in software and hardware were explored. Using the Python language and Django as a development framework, the next version of the simulator was built to exploit the high-performance computing capabilities for research with a web interface. Next, using a modular paradigm, a second version of a simulator was made based on object-oriented programming (OOP) having "high-cohesion", "low-coupling", and extensibility features. A third numerical version with some libraries like NumPy was created and tested, looking for better performance. All these versions served as a point of departure to explore the basis of a fourth and fifth version: a high-performance simulator with a parallel mindset, using Graphic Processing Units (GPUs). A comparison is presented with the same single cross example testing the time to generate data in each version.

1 | Introduction

The world population has grown rapidly in recent years. According to some forecasts [1], there will be about 9.73 billion inhabitants worldwide by 2050. This poses many problems, related to the mobility of people, physical and mental health, as well as issues related to their impact on nature. More specifically, in urban areas, mobility is a major issue where the concentration of people is high. Some studies such as [2] indicate that in some cities people can spend up to 272 hours a year stuck in traffic.

Even when people use a variety of transportation options, cars and public transport are still some of the most widely used. This requires effort to overcome traffic due to the size of roads, the infrastructure available, and the number of vehicles. So, the study of Vehicle Traffic Networks (VTNs) is still an open subject

[3, 4] where multiple factors such as congestion, driver's habits, and heterogeneity on the roads are active in most of the networks studied [5].

VTNs have been studied exploring different approaches. A helpful review was published in [6]. Typically, the study of VTNs can be differentiated in two main focuses: Microscopic and macroscopic [7, 8]. Microscopic-based works design the elements in VTNs with specific details. For example, in some cases, cars are represented with tiny details like speed, size, and acceleration; also, drivers are studied considering their habits [9–11]. In contrast, macroscopic perspective represents VTNs at a wider level, in which vehicular traffic is conceived regularly as a flow. In this case, flow density is the main parameter to define state space models in discrete or continuous time [12]. Alternative models have been studied like in [13] based on energy. Congestion modeling is addressed in [14]. A disadvantage of this approach is the loss of information due to the focus on the flow density [15]. In an intermediate hybrid level, we can find the mesoscopic perspective, with a mix between the two major alternatives. In this perspective, the flow usually is depicted in groups of cars [16]. Finally, as shown in [17, 18], the traffic models serve to enable autonomous driving and other further applications.

The main purpose of Multiagent Dynamical Systems (MAS) is the representation of complex systems by the interconnection of agents [19]. An important challenge of MAS is to design the connectivity of different agents and the huge amount of information shared between them [20]. The interaction of agents can be represented using different approaches, for instance, discrete event mechanisms, differential equations and probability. Different applications have come out from the MAS, as optimizing routes, coordinating robot movement, managing large economic systems and planning project tasks [21]. Another example of agents' approach is the one proposed in [22] where a complex system has been split in agents, using the design of algorithms to solve the Adaptive Dynamic Programming (ADP).

Control traffic mechanisms have the objective to fetch data to improve vehicular flow synchronously, analyzing data in real-time and making changes in real-time, or asynchronously, using tools to analyze information and define plans and strategies which can be implemented and tested after their design. Sensors are used to obtain information to be analyzed. In some cases, sensors are installed on roads or in traffic lights to measure and control vehicular traffic. In other cases, like [4], some smartphones of motorists or low-cost GPS vehicular embedded sensors are compared with sensors deployed in the existing vehicular infrastructure to validate the models in real-time. The amount of data and the complexity of working with VTNs require specific computational resources that can be a good fit in obtaining timely and accurate information. Therefore, computer simulators have historically helped in the research and analysis of VTNs.

Some microscopic commercial and non-commercial simulators [23–25] are: Traffic Software Integrated System, CORridor SIMulation (TSIS-CORSIM), AIMSUN, and Simulation of Urban Mobility (SUMO). These simulators work at a microscopical level representing cars on different roads and paths displayed on the graphical interface in simulators. The consequence of the microscopic viewpoint is the growth of the VTN as the number of elements to be processed and displayed on the screen increases. It requires a high computational cost making the study of larger grids difficult. As expected, some intersections include traffic lights ordering the movement of cars. These kinds of crossings can be addressed using different representations. The most common is a discrete-event option called Petri Nets (PN). The temporal evolution of tokens in a PN can describe the traffic light cycle place. Some relevant PN cases are presented in [26–33]. In this paper, we use a simple timed state machine to represent the functioning of traffic lights as alternative in our previous works [34–36].

On the other hand, computational traffic simulator tools have historically helped in the understanding, analysis and the improvement of the flow in VTNs. Considering the huge amount of information captured from the number of cars mainly in crowded cities and urban spaces, some systems are used to solve this complexity. That is the case of [34] where a tool was designed to model and simulate clusters of 1,000 nodes and traffic agents using a Central Process Unit (CPU) in a distributed and concurrent way. Specifically [35], works with a model of 120 million elements, represented in an agent-based model, with data from different firms settled in the USA using a parallel simulator.

Some articles have worked on traffic simulators related to car collisions, such as [36], which include a methodology to study crashes and their consequences, considering potential mistakes made by drivers and obstacles and their trajectories. The case of [37] looks to create a framework to assess potential risks in roads such as crashes or adverse weather. Some cases like [38–40] have simulated the behavior of autonomous cars based on different factors like interaction with cars driven by people and the self-driving psychology. An aspect widely studied in agent networks is the control and automation of connections in interagent connections; for instance, the work of [41] introduces an adaptive intersection.

Some other works [42] explore systems of monitoring and processing unmanned aerial vehicles transit. The proposed system achieved an 88% average success rate in real-time vehicle detection, incorporating a high-performance neural accelerator which is a relevant study case for traffic research and, in specific, for this work.

This make evident that the HPC systems will be crucial in the development of strategies for transit control, optimization and policies development. With the grown of different type of vehicles, including those that use aerial space will represent a computational challenge where the flow and movement of them will represent an interesting challenge to be addressed.

The main objective of this work is to explore the different possibilities to evolve the traffic simulator in a computational sense. We explored different options not only in software but also in hardware. Then, in this work, we analyze the alternatives to show the route we followed from a MATLAB-based simulator to a new version of a simulator based on high-performance hardware. We evolved the simulator considering some hardware and software options, aiming to improve information retrieval and thereby defining the best structure in a VTN that allows a better traffic flow.

The essential contributions of this work are summarized as follows:

- Continuing with our prior work in [43], we have started the exploration of some compute alternatives to be prepared towards having a more scalable solution.

- Based on the initial model presented in [43], running in MATLAB, we started to check possibilities to evolve into a more capable version, taking advantage of the latest computational advancements.
- Show the experiences and challenges we faced when trying to evolve a MATLAB simulation to a more robust simulation tool
- Different simulator options are explored to test whether a better performance can be achieved and, consequently, timely information and responses can be obtained.
- An object-oriented programming (OOP) approach is used to match with the existing modular traffic model.
- A first reflection about the overhead introduced by this type of approach is shown in our work.
- Considering a long-term simulator designed for end users, a new web-based framework was integrated.
- Two calculation bases were explored on Python, first with an OOP basis and then with a numerical version based on the differential equation of the whole VTN.
- We showed the difference between these versions and a possible explanation of the observed behavior.
- A translation of the simulator was conducted going from a serial mindset on CPU to a parallel version on GPU, considering the algorithm designs and the coding. Insights, experiences, and metrics about this evolution are shown in this work.
- Considering a single cross as a base, we defined a set of parameters to be run in each version of the simulator. First, we assess the correctness of the data generated, and second, we measure the performance by checking the time the simulator spends to generate the data.
- We explored different configurations in execution parameters in GPU, trying to achieve a first version of high-performance computing.

2 | Materials and Methods

We performed our tests with the same parameters in each version, using a workstation with an Intel Xeon W2245 3.0 GHz processor, 128 GB RAM, GPU NVIDIA Quadro RTX 8000 (Turing) 48 GB RAM, Ubuntu OS. We used Python as the backend programming language, Django as the web framework, Atom as the selected Integrated Development Environment; and Git for version control.

Each version in the simulator is managed by a view controller in the Django framework; the variables of the execution are received as parameters in each view in different Python files. Talking about GPU, the CPU served as the input method to introduce the values to the GPU, passing numerical values through NumPy arrays with Numba as the chosen library.

The frontend has been coded using the standard languages for these purposes, like HTML and CSS files.

The model used in this article to perform the simulations in the different platforms or languages is based on the model introduced in [43], where a new proposal for modeling complex networks of vehicular flow using simple differential equations was presented, with a multi-agent systems approach at a macroscopic level of detail. These agents have information on change in flow, density and changes in traffic signals. The model carried out in [43] was compared with simulations carried out in TSIS-CORSIM, with real vehicle density data, achieving comparable results, considering the differences in the level of detail with respect to commercial software, finally highlighting that the [43] model allows the simulation of complex networks with fewer computational resources.

3 | Results and Discussion

3.1 | Agent-Based Modeling of a VTN

This section presents a brief summary of the modeling of VTNs presented in [44]. A VTN can be conceived as a set of interconnected dynamical agents representing street segments. The set of n physical spaces with flow of vehicles is given by $N = \{\beta_1, \ldots, \beta_n\}$. A VTN is developed as a connectivity graph in which vehicles go from an agent to a subsequent one. Then, N_i^+ is the inadjacent subset of N and N_i^- be the out-adjacent subset of N. When $N_i^+, N_i^- \neq \varnothing$, it is possible to partition the set into two subsets, the N_{ST} subset which corresponds to streets, and the subset N_{CR} which corresponds to intersections. Within the subset N_{CR} there could be a subset of intersections with traffic lights denoted by N_{TL}.

The nonnegative occupancy of each agent β_i, given by z_i, is represented by the following first-order differential equation.

$$\dot{z}_i = \sum_{j \in N_i^+} u_{ji} - \sum_{k \in N_i^-} y_{ik}, \tag{1}$$

$$y_{ik} = \sigma_{ik}(z_i, z_k) z_i, \qquad \forall k \in N- \tag{2}$$

Equation (1) shows the balance of inputs and outputs in each segment. The occupancy is delimited by $0 \leq z_i \leq z_i^{\max}$, where $z_i^{\max} \in \Re^+$ is defined according to the physical space of β_i. Input u_{ji} is the flow entering of the street segment of β_i. Equation (2) illustrates the behavior of the output flow in β_i. Output y_{ik} is a discharge flow function conceived as a time-variant nonlinear discharge coefficient. It is evident that the value of $\sigma_{ik} = 0$ describes the flow, stopping it or increasing according to the case. The model allows that agents can be connected in a modular way to develop a small or a big VTN, regardless the size or shape, this ability in conjunction with $a_k(z_k)$, $\alpha_i(z_i)$, and Γ_{ik} values are illustrated in Figure 1.

The parameter σ_{ik} computes the combination of three main factors: self-congestion through $\alpha_i(z_i)$, the congestion in subsequent segment through $a_k(z_k)$, and Traffic Light or Flow-Distribution (TL-FD) effects contained in Γ_{ik}. Thus, σ_{ik} is defined as

$$\sigma_{ik}(z_i, z_k) = \alpha_i(z_i) a_k(z_k) \Gamma_{ik} \tag{3}$$

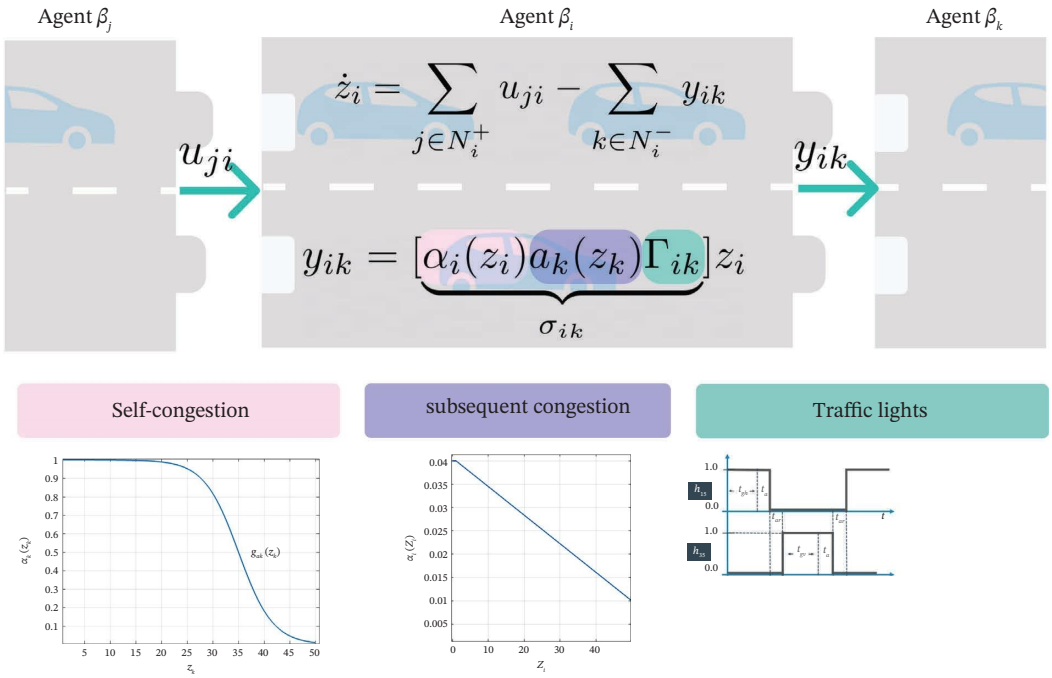

FIGURE 1 | Condensed model.

The functions $\alpha_i(z_i)$ and $a_k(z_k)$ are defined as shown in Equations (4) and (5).

$$\alpha_i(z_i) = \begin{cases} \alpha_i^{\max} & \text{if } 0 \le z_i \le \rho_i, \\ f_{\alpha_i}(z_i) & \text{if } \rho_i < z_i < \mu_i, \\ \alpha_i^{\min} & \text{if } \mu_i \le z_i \le z_i^{\max}, \end{cases} \tag{4}$$

$$a_k(z_k) = \begin{cases} 1 & \text{if } z_k = 0, \\ g_{a_k}(z_k) & \text{if } 0 < z_k < z_k^{\max}, \\ 0 & \text{if } z_k \ge z_k^{\max}, \end{cases} \tag{5}$$

where the functions $f_{\alpha_i}(z_i)$ and $g_{a_k}(z_k)$ are smooth functions given by:

$$\mathrm{f}_{\alpha i}(\mathrm{z_i}) = \alpha_{\mathrm{i}}^{\max} - (z_i - \rho_i)\frac{(\alpha_{\mathrm{i}}^{\max} - \alpha_{\mathrm{i}}^{\min})}{\mu \mathrm{i} - \rho \mathrm{i}}, \tag{6}$$

$$g_{a_k}(z_k) = \frac{1}{1 + \exp\left(\lambda_k\left(\left((z_k)/(z_k^{\max})\right) - \xi_k\right)\right)}. \tag{7}$$

The parameters $0 \le \rho_i < \mu_i \le z_i^{\max}$ serve as the occupancy interval where the function $\alpha_i(z_i)$ decreases. Also, the parameters $\lambda_k > 0$ and $0 < \xi_k < 1$ define the shape of the sigmoidal function that gives the percentage of flow going through to downstream agent β_k.

Figure 2 shows an example of $\alpha_i(z_i)$. Figure 3 shows an example of an $a_k(z_k)$ function.

Finally, the function representing Traffic Lights is described by Γ_{ik} which depends on different cases if β_i and the subsequent β_k belong or not to N_{CR} or N_{TL} as given in equation (8).

$$\Gamma_{ik} = \begin{cases} h_{ik}, & \text{if } \beta_i \notin N_{CR}, \beta_k \in N_{TL}, \\ w_{ik}, & \text{if } \beta_i \in N_{TL}, \beta_k \notin N_{CR}, \\ 1, & \text{otherwise.} \end{cases} \tag{8}$$

with $h_{ik} \in \{0, 1\}$ and it changes value depending on the stage of the finite-state machine which is attached to traffic light cycle in each β_k. The weighing value $w_{ik} \in \Re, 0 < w_{ik} < 1$ contains the proportional flow in each intersection. The cars in $\beta_i \in N_{TL}$ are fully distributed in agents $\beta_k \in N_i^-$, then it is satisfied that $\sum_{k \in N_i^-} w_{ik} = 1 . \Gamma_{ik} = 1$ by default when traffic lights are not involved in the intersection allowing a complete flow of vehicles in that direction.

Figure 4 illustrates $\beta_i \in N_{TL}$. When h_{ji} enables or stops the output from β_j to β_i and w_{ik} indicate the proportion of flow leaving from β_i to β_k.

The value h_{ji} defines the traffic signal to each connection between the agents, and it depends on the traffic lights' cycles. These cycles serve to enable green, yellow and red lights, like a common traffic light. The behavior of these machines can be simulated using some discrete-event techniques such as timed PNs and timed finite-state machines, among other discrete structures.

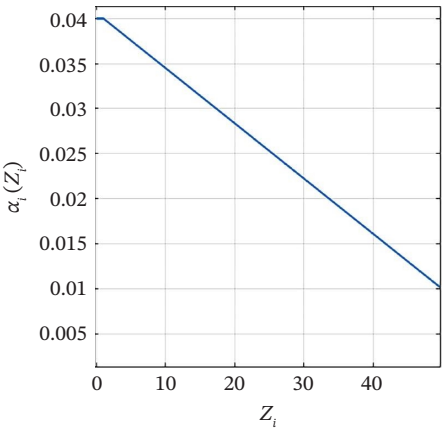

FIGURE 2 | Self-congestion function $\alpha_i(z_i)$ value.

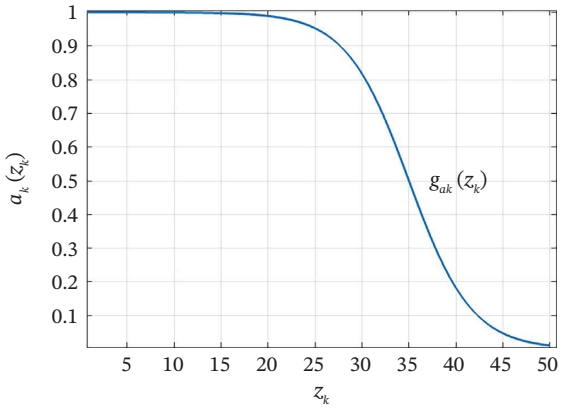

FIGURE 3 | Congestion in subsequent segment function a_k.

Now, to extend the model and look for a more reduced version of the equations, we used a matrix representation, which aids in the simulation and serves as a future control structure. The connectivity of agents establishes that if $j \in N_i^-, i \in N_j^+$. Meanwhile $\Gamma_{ij} = \Gamma_{ji}, \forall i \neq j$. Then, the adjacency matrix $M \in \Re^{n\times n}$ with elements $m_{ij}, i = 1, \ldots, n, j = 1, \ldots, n$ defined in equation (9).

$$m_{ij} = \begin{cases} \Gamma_{ij}, & \text{if } \beta_j \in N_i^-, \\ \Gamma_{ji}, & \text{if } \beta_j \in N_i^+, \\ 0, & \text{otherwise.} \end{cases} \tag{9}$$

The out-adjacency matrix $M^- \in \Re^{n\times n}$ with elements $m_{ij}^-, i = 1, .., n, j = 1, \ldots, n$ in equation (10).

$$m_{ij}^- = \begin{cases} 1, & \text{if } \beta_j \in N_i^-, \\ 0, & \text{otherwise.} \end{cases} \tag{10}$$

And consider the matrix equations.

$$z = [z_1, \ldots, z_n]^\top, \tag{11}$$

$$a = [a_1(z_1), \ldots, a_n(z_n)]^\top, \tag{12}$$

$$\boldsymbol{Z}_d = \text{diag}[z_1, \ldots, z_n], \tag{13}$$

$$\boldsymbol{\alpha} = \text{diag}[a_1(z_1), \ldots, a_n(z_n)], \tag{14}$$

$$\boldsymbol{A} = \text{diag}[a_1(z_1), \ldots, a_n(z_n)]. \tag{15}$$

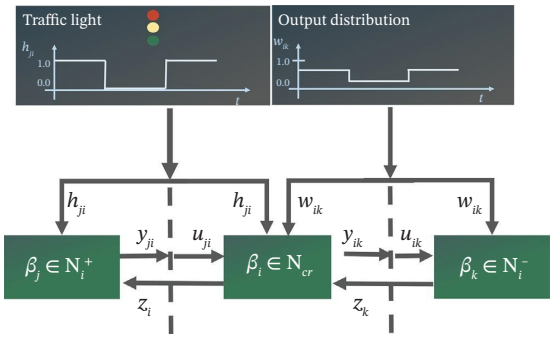

FIGURE 4 | Block diagram of an intersection with traffic light.

Then, the next matrix equations [45] are equivalent, representing the equations of the systems given in (1)-(2) for a VTN with n agents:

$$\dot{z} = AB_T\alpha z - (\alpha Z_d B)a, \tag{16}$$

$$\dot{z} = \left\{AB^\top \alpha Z_d - \alpha Z_d BA\right\}\overrightarrow{1}, \tag{17}$$

$$\begin{aligned} \dot{z} = \{&\left[-BA(M^-)^\top\alpha\right]I_n \circ +, \\ &B_T \circ \left[A(M^-)^\top\alpha\right]\}z, \end{aligned} \tag{18}$$

where $B = M \circ M^-$ and $B_T = M \circ (M^-)^\top$, with $\circ$ representing Hadamard multiplication [45], (elementwise multiplication), $\overrightarrow{1} = [1, \ldots, 1]^\top \in \Re^{n\times 1}$ and I_n the $n \times n$ identity matrix.

3.2 | Example of VTN

Considering Downtown in Mexico City as the base (Figure 5), we simulate a simple crossing example. In this example $N = 9$, where $\beta_1, \ldots, \beta_4$ are streets, β_6, β_7 are sources, β_8, β_9 are sinks and $\beta_5 \in N_{TL}$ is a junction with a traffic light. A west-to-east flow in one direction is represented by $\beta_6 \longrightarrow \beta_1 \longrightarrow \beta_5 \longrightarrow \beta_2 \longrightarrow \beta_8$, and north-to-south flow is represented by $\beta_7 \longrightarrow \beta_3 \longrightarrow \beta_5 \longrightarrow \beta_4 \longrightarrow \beta_9$. The traffic light cycle controls the flow between both directions.

The traffic light performs its cycle to allow the flow in which direction, alternating the west-to-east flow and north-to-south flow, assigning green-time values t_{g_h} and t_{g_v}, respectively, t_a is yellow time and t_{ar} the all-red time. In Figure 6 values of $h_{15}, h_{35}, w_{52}, w_{54}$ are presented according with t_{g_h}, t_{g_v}, t_a time values. The period in each phase is calculated by $T = t_{g_h} + t_{g_v} + 2(t_a + t_{ar})$. In the horizontal direction $w_{52} = 0.6, w_{54} = 0.4$, 60 % of vehicles move horizontally and 40 % turn right to the vertical flow, while in the vertical direction they change to $w_{52} = 0.4, w_{54} = 0.6$, 40 % go straight vertically and the rest of 60 % turn left. The phases at traffic light 6 are created by a finite-state machine which is shown in Figure 7. States 1-2 and

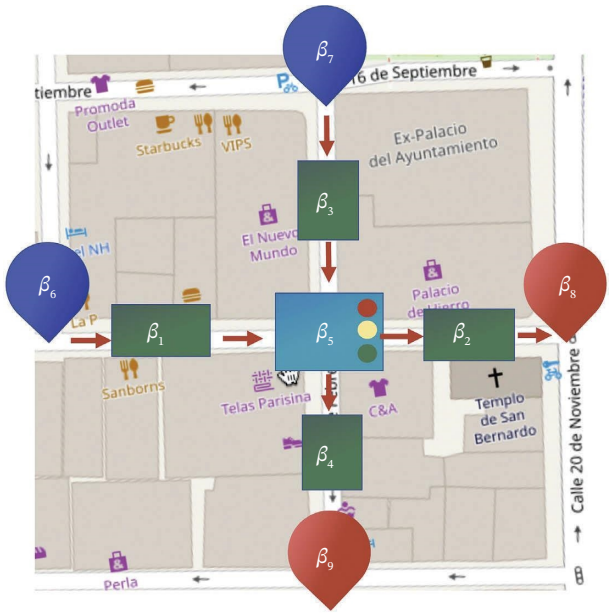

FIGURE 5 | Example of VTN.

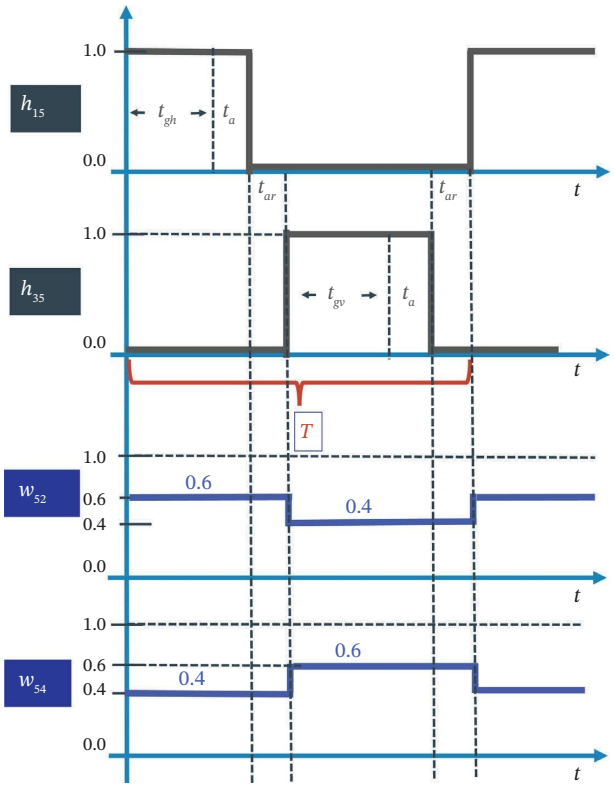

FIGURE 6 | Signal generator for h_{15}, h_{35}, w_{52}, and w_{54}.

3-4 correspond to each phase, respectively. The transition at each node is given by green, yellow, and all-red times. Values h_{15}, h_{35}, w_{52}, w_{54} are indicated in every state and are used in the simulator to enable/disable the flow depending on the case.

3.3 | Simulators of the VTN Model

The previous agent-based modeling of a VTN can be simulated by different computational languages and platforms. Note that every differential equation is simple to program, but the main challenge is running the entire model with a significant number of streets and intersections that appear in real cases. Therefore, this section presents the different versions of simulators built for the model. The analysis and comparison of these versions are an important step for the deployment for the final optimized version.

An overview of simulators timeline is shown in Figure 8.

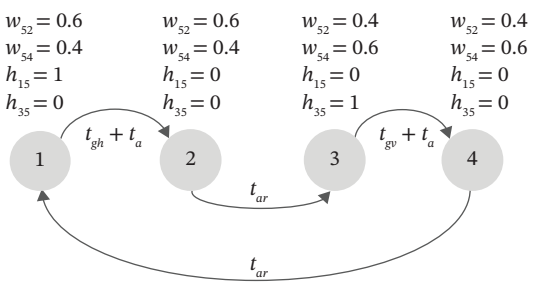

FIGURE 7 | Example of timed finite-state machine.

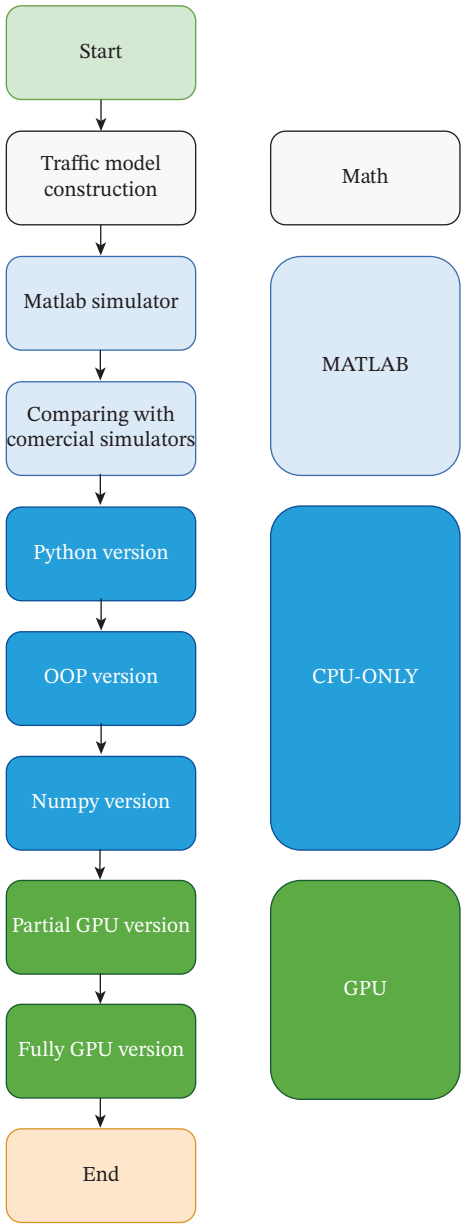

FIGURE 8 | Simulators timeline.

3.4 | First Simulator Version

We started with the MATLAB simulator presented in [43, 44], with a graphical and coded version. The graphical version was created using Simulink and the coded version using an m-file, or script file [46]. Figure 9 shows an image of the model in Simulink, while Figure 10 presents a block version of the algorithm.

Several arrays and variables were configured to parameterize the simulations; values like $\alpha_i(z_i)$ are used to simulate self-congestion and $a_k(z_k)$ to define subsequent-congestion.

Then, Γ_{ik} values to depict PN cycles were configured. Initial z_i and max $z_i^{\max}$ values of each street segment were defined. The representation of inputs and outputs G in the simulator was originally produced using variables managed separately; nevertheless, having in mind faster calculations, a matrix approach was adopted for all the variables, and they could be available when they would be relevant.

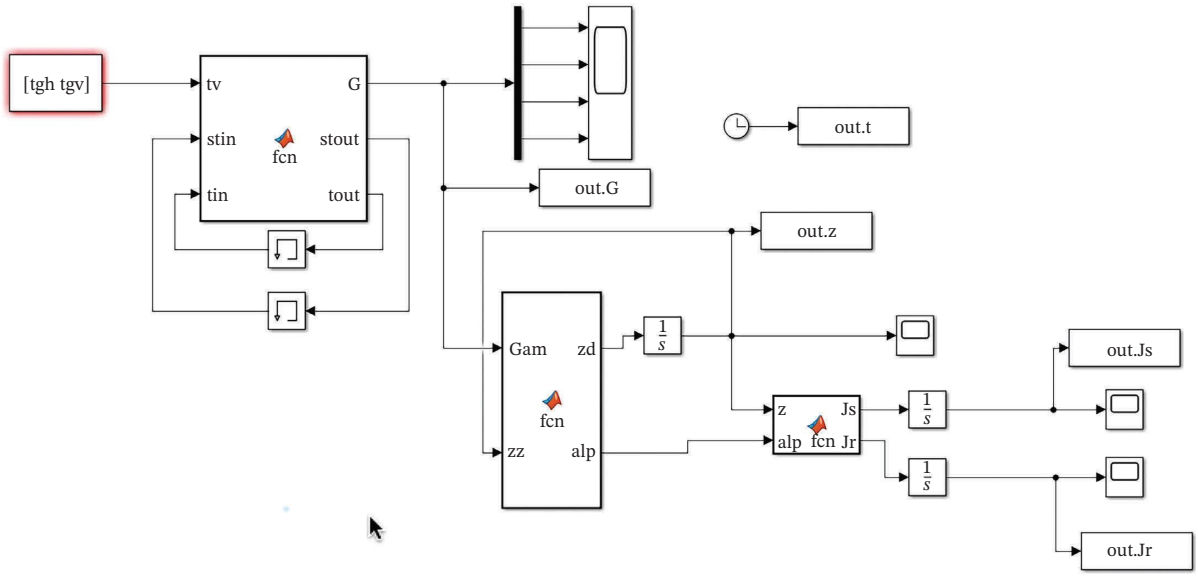

FIGURE 9 | Simulink design.

The matrix representation helped the programming of equations in the model. Thus, it enables the system's structure for better schemes to improve the flow of vehicles. To reach this, an adjacency matrix was created.

Next, elements like $f_{\alpha_i}(z_i)$ were embedded into functions in MATLAB.

The first simulations have been conducted using PN to represent traffic signals, then a simple finite-state machine presented in Figure 7 was defined. The different states depict the cycles of the traffic signals depending on the green, yellow, and all-red times.

Finally, cycles and vars were established in MATLAB functions to process the model. Results can be found in our previous works. Some main aspects that enabled us to convert the simulator to a different platform are outlined next:

Having a code version was fundamental to create a new programming version because Simulink is a graphical tool available only for MATLAB and not available for other languages.

- Going from a PN version to a finite-state machine was central due to the ease of its integration into a parallel cycle.
- Having matrix calculations with structures oriented to calculations facilitated the integration of performance measurement and improvement in the new version.
- Having a clear method of integration was fundamental for coding a simulator with similar results.

Finally, in this initial version, all the simulations were conducted using primarily MATLAB in two different forms: in a Simulink fashion and in a coding fashion. These two methods served to verify the model and confirm if we would have the same results in data. For more information, the conclusions of using the MATLAB simulator are published in [43, 47, 48].

3.4.1 | The Second Version of Simulator

Using MATLAB version as a starting point, we selected Python as a new language to code the simulator. Python is a general-purpose programming language and is known for its simplicity and readability. It is an opensource, multiplatform language used in domains related to data science, machine learning, and high-performance computing with NumPy and Numba. For those reasons, this language emerged as the primary selection.

For an end-user system, it is crucial to present an interface where the user can operate. Considering the requisite that the framework had to be compatible with Python. Then Django emerged as a well-documented and suitable option.

Django [49] is an appealing and well-known framework that integrates Python with the opportunity to display a webpage. According to the official documentation [50], implementing Django, you can take web applications from concept to launch quickly. Django takes care of much of the hassle of web development, so you can focus on writing your app (in the simulator,

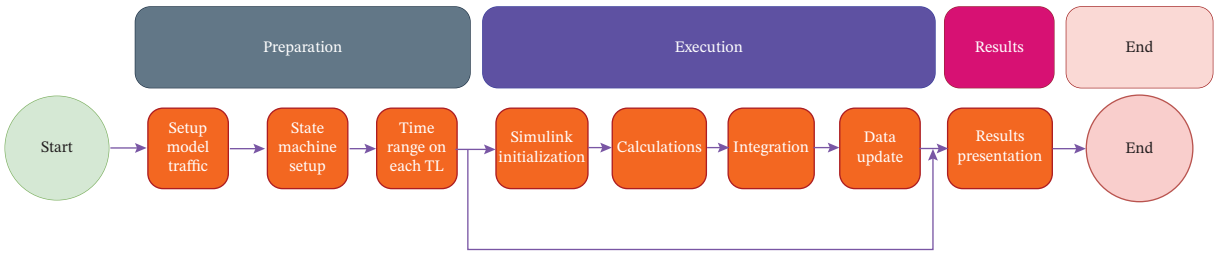

FIGURE 10 | Blocks diagram of the MATLAB version.

this is vital because it is possible to focus on improving the performance).

Using a virtual environment, Django set up a framework to build a webpage with all the necessary attributes, such as the possibility of viewing it in a browser, connecting with other web services, mounting it in a hosting service, to enable the option to interact with users through a browser and some others. Figures 11 and 12 show the preliminary UX/UI (User eXperience, User Interface) of the landing page and simulator view.

In this preliminary UX/UI interface, multiple features were designed. Some of them are still in progress; however, having a complete design serves as a reference and roadmap for the simulator we are developing.

First, we created a landing page where users can start interacting. Later, the simulator main page is shown. On this page the user can select from three different options (processors):

- CPU—CPU for input and simulation tasks
- CPU-GPU—CPU for input and output; GPU for processing
- CPU-GPU-QPU—CPU for input and output; GPU for processing; Quantum Process Unit for optimization (future work)

Note that the CPU is present in all choices because this kind of processor cannot be replaced as fundamental unit, in our case as the main media to interact with user.

Once the processor is selected, the user will frame the map, and the algorithm will identify traffic lights on it, and then the user could enter different parameters to explore and simulate scenarios.

Simulation performs its tasks and exposes results depending on criteria selected.

The goals include ensuring that the simulator can work with VTN information in a real sense, that is, acquiring data from existing streets, roads, avenues, and crossings. Considering this, some analysis of commercial simulators and how they handle data was performed. The usual structure to represent a VTN is considering a set of points (nodes) and the connection between them as edges to represent roads.

OpenStreetMap [51] provides a free API to obtain maps from the real world. To detect the information attached to a map selection

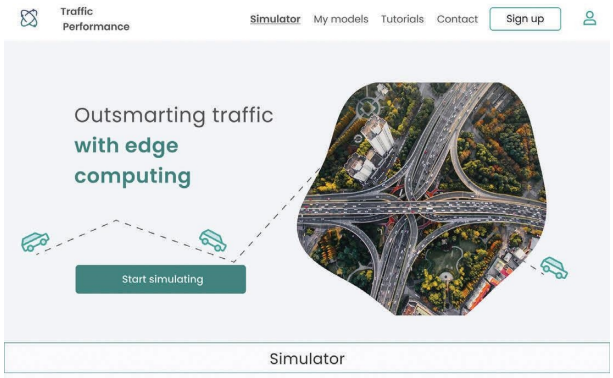

FIGURE 11 | Main screen.

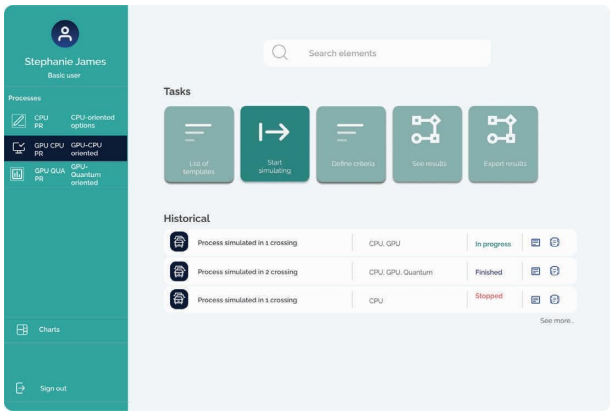

FIGURE 12 | Simulator view.

in the interface of the simulator, we selected OpenStreetMap as the option to acquire data about nodes and edges for the real world.

We started studying a single intersection which can be seen in Figure 13 using the agent-based modeling presented in the previous section. Streets are represented by gray pins and intersections by blue pins.

An example of a bigger representation of the VTN considering a larger section of Mexico City is shown in Figure 14.

A possible solution when trying to build a scalable, modular and repeatable program is to use classes. Classes are a concept formulated in OOP because it allows developers to reduce coding abstraction, getting closer to how the world that is perceived by humans and reusing code. Usually, we try to organize elements in the world like Classes, first, then when they become more "tangible" and "executable", in Objects. Languages like Java and C++ popularized the term.

Talking about traffic, some works have already used OOP to simulate traffic with a significant number of agents, for example [35], created a model of agents using OOP reaching up to 120 million agents on a parallel environment using the CPU. Consequently, analyzing the objects, having streets, intersections, among other components, the logical idea was to convert the model into an OOP paradigm.

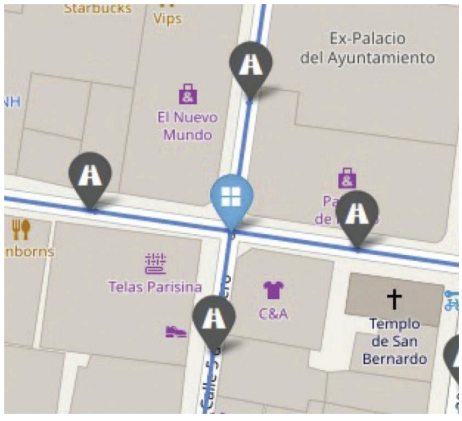

FIGURE 13 | Single intersection using OpenStreetMap.

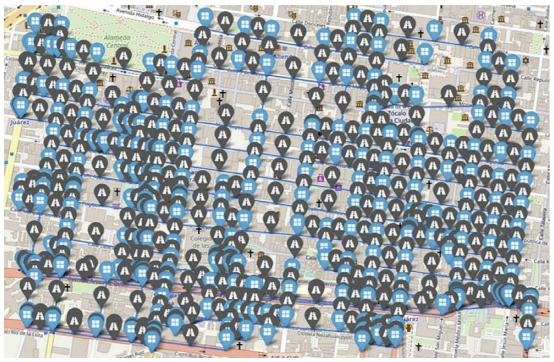

FIGURE 14 | Bigger selection using OpenStreetMap showing a massive number of agents.

Considering elements in the real world, we first created a Class including characteristics and actions in an OOP mindset, like properties and methods. Then, setting the type of data involved in each property, and secondly a method called constructor to initialize streetSegments.

This constructor serves to create objects setting properties and methods of the class, i.e. the implementation of this class is an object that can be used in the simulation. A representation of the route from real world to the class abstraction and ending in implementation in objects can be seen in Figure 15.

Each structure of a street is modelled first in a Class and next, using a constructor (term used commonly in OOP for methods which serves to create objects), streetSegment(), and object (in pink color), is built with all properties obtained from its class. As we can see the abstraction using Class matches the description of the agents.

Considering the modularity in the model and the fact that the connection between streets happens in a massive way, OOP represented a good option to explore about. In a larger street representation, a single crossing is shown in Figure 16. As we can see, a crossing can be divided into five different objects. Using the same method mentioned above, the class is used through its constructor to generate five different objects.

Each object connects itself with other objects through their properties used a similar approach to those used in database models. In Figure 17 a block diagram of this version is shown.

3.4.2 | Third Version of Simulator

The performance was negatively impacted considering pure OOP, as we will see in section Results. This may be since OOP read/write in objects is slower than having access to a simpler data structure, such as arrays. Therefore, a new strategy was adopted to improve the performance in the simulator.

As a main study in this work, we only performed a time comparison between simulations. Nevertheless, trying to have a response for this performance reduction, we researched more about these structures in Python.

This task was not trivial because common language documentation focuses on the use of the language.

CPython [52] is the basis of modern Python, which usually defines how Python behaves in terms of performance, memory management, internal structures, and so on.

Looking more in detail at Python module, class, and object specs, we detected some layers (not too heavy) of tuples that control the usual functionalities of the OOP paradigm, like indexing, tuples of dictionaries saving keys and values, definition of type of instance, and a collection of possible interwoven hierarchies that deal with aspects such as inheritance and polymorphism.

Without stopping considering OOP as an option, we worked in some modifications, focusing more on a numerical simulation using the available tools in Python. The first answer to emerge was from NumPy [53]. NumPy is "The fundamental package for scientific computing with Python" according with the official documentation. To achieve this, having an array-based simulator has been crucial to have the code version with a set of matrices created previously in the model.

One additional issue was the integration method. Even though similar integration methods exist in Python, the election of the method that adjusts the algorithm developed from a MATLAB

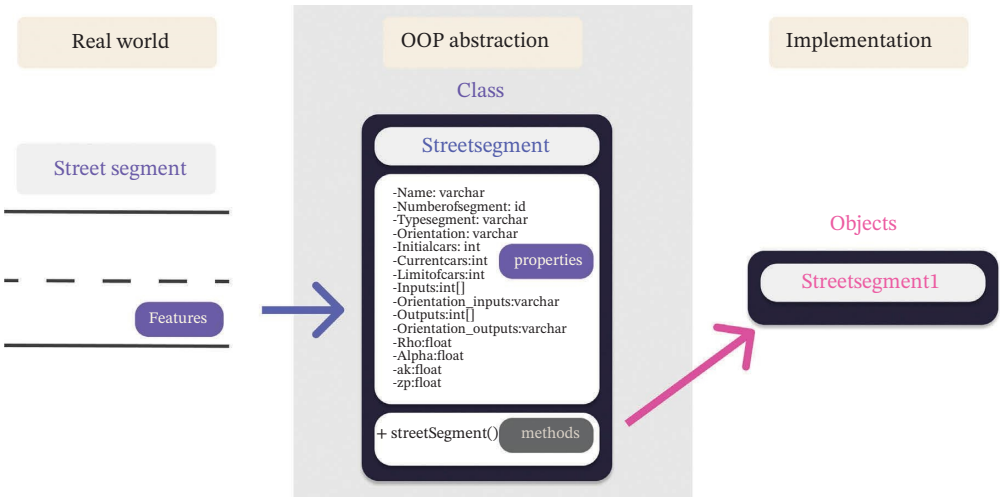

FIGURE 15 | OOP single street representation.

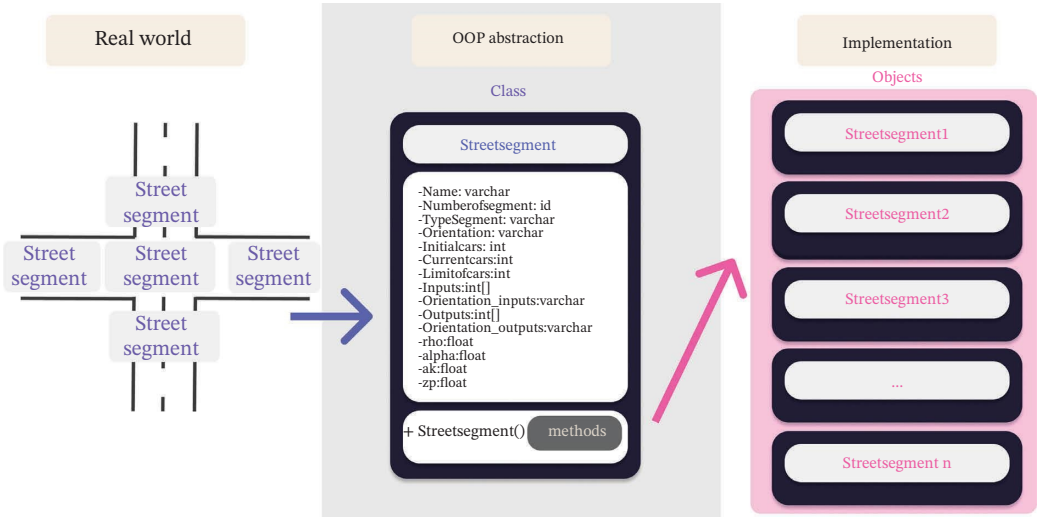

FIGURE 16 | OOP crossing representation.

version required exploration and conducting different experiments to find a suitable method in Python.

A block diagram of this version is shown in Figure 18 and the processing steps in Algorithm 1.

The first condition was to produce the same results with the new version of the simulator compared with the baseline simulator built in MATLAB; then, in terms of integration, the option that provided an accurate result was the odeint function accessed via SciPy library from Python [54].

Considering that the GPU is a device that enters the equation in general-purpose computing, it comes with different limitations and benefits according with its inherent architecture.

For GPU, there are multiple operations commonly present in CPU from decades, but in the case of GPU, due to its initial purposes and then of its evolution, are not present.

Some of them are due to the initial purposes of the creation of GPUs, and some are related to the evolution of these devices steering to AI tasks.

References [55, 56] detailed that the answer lies in the differences in fundamental design philosophies between the two types of processors. The design of a CPU is optimized for low latency and sequential code execution, while the GPU is optimized for parallel tasks.

The integration method is not currently supported natively on Numba with Python; for that reason, we developed the equations in GPU-kernel functions (only available from other GPU-kernels). This development represented an important technical challenge, because even when arithmetic and algebraic operations are common in GPU too, we could not use odeint natively.

Another reason is that even when some operations are available in GPU, not all the operations are optimized like what happens with the CPU.

Then, we had to understand how MATLAB and Python perform their calculations and develop the methods in GPU, having in mind that the results were equal. The help of the math team here was crucial to achieve the same results.

We confirmed that the values across versions do not show significant differences in terms of the project scope. This was a careful process, taking into account that integrations between each processing step are more visible in cumulative procedures, where even small differences can accumulate and become more evident.

In order to generate a comparison in computational complexity between CPU and GPU versions both working with NumPy arrays, a diagram of the algorithm of this version was generated, shown in Algorithm 1, where β is the selected VTN represented as segments of streets, $tl1_{\min}$, $tl1_{\max}$, $tl2_{\min}$, $tl2_{\max}$ depict the parameters of each traffic light, in this case for a single crossing; and lastly z for the number of vehicles initially represented as a scalar in each β and later measured as flow in z_p.

3.4.3 | Fourth Version of Simulator

Now, once we had reached a numerical array-oriented version, we started exploring options to speed up the performance while preserving simulation correctness. Considering the latest

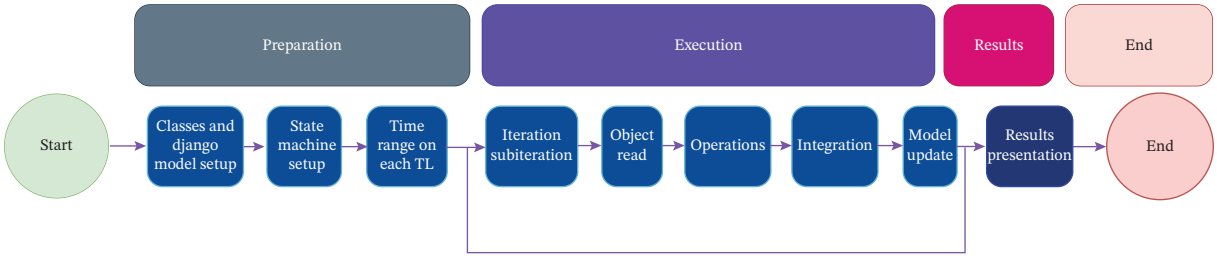

FIGURE 17 | Blocks diagram of the object-oriented programming version.

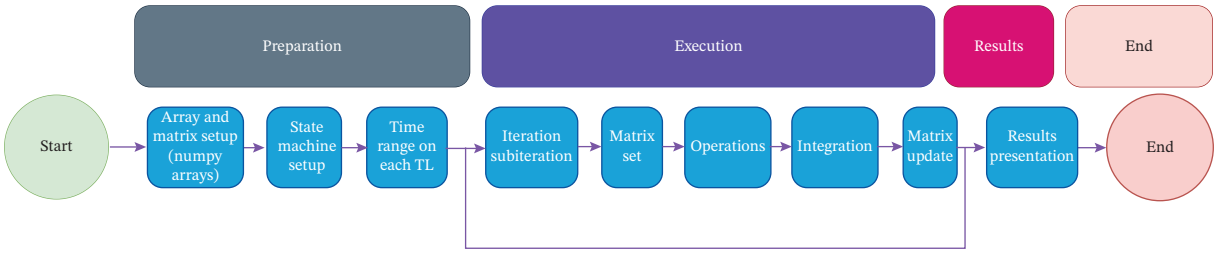

FIGURE 18 | Blocks diagram of the NumPy version.

ALGORITHM 1 | CPU NumPy algorithm.

Input β, $tl1_{\min}$, $tl1_{\max}$, $tl2_{\min}$, $tl2_{\max}$, z_0

Output z_p

$simulations \leftarrow (tl1_{\max} - tl1_{\min}) * (tl2_{\max} - tl2_{\min})$

for n in *simulations* **do**

  Calculate z_p

  Integrate z_p

  $n \leftarrow n + 1$

end for

advancements in parallelized architectures and high-performance computing, we identified that GPU computing is applicable to speed up processing [57–60] and, in particular, to accelerate the simulation of vehicular traffic [61–67].

Several years ago, GPUs were first built for videogames to accelerate specific 3D rendering tasks. The early architecture was designed to draw millions of elements in videogames rendering them on real-time platforms. This architecture favored multi-processing over memory latency commonly found in CPUs. GPUs are commonly used to perform massive tasks, as it allows to execute them in a parallel way that is explained because GPUs have a higher number of Arithmetic Logical Units (ALU) than CPUs.

A general representation of CPUs and GPUs is shown in Figure 19, where it is clear that the GPU is a processor that is made up of many smaller and more specialized cores focused on throughput behavior, whereas the CPU has an architecture oriented to latency. CPUs are mainly focused on general processing; for that reason, latency is crucial to enable users to execute and access tasks with different purposes. In the case of GPUs, the architecture is more defined to process massive calculations at a time. By working together with CPU, the cores deliver massive performance when a processing task can be divided and processed across many cores [68].

Then, some years ago, the scientific and professional community realized that GPUs could be used more broadly to enhance computing capabilities and reach a better performance running massive processes. To gain better understanding of GPUs and the parallel computing mindset, we had to check for documentation and resources to understand how GPUs operate and then create an algorithm to execute the simulator in a parallel fashion using GPUs. One resource consulted has been the teaching kits for high-performance computing which comes from an alliance with NVIDIA, and the second was a certification in fundamentals of high-performance computing using Python. Once we had gotten these, a more detailed understanding of how GPUs operate through kernels that are executed in a parallel manner was possible. The execution is run by segments or chunks in memory, a processing structure in grids, blocks, and threads.

Figure 20 illustrates this internal arrangement on GPU devices.

GPUs achieve better performance when they are used to conduct numerical processing functions with thousands or millions of operations. When using less data, there is a computational expense due to the overhead necessary to operate GPUs which is greater than the acceleration achieved. Then, this overhead in GPU processing is only convenient when processing massive operations (thousands to millions).

Another important concept is that, from the field of high-performance computing, CPUs take the name of the "host" and GPU the "device". This terminology is crucial because the data transfer between both devices during usage is very common, so having a well-defined nomenclature and a correct handling of

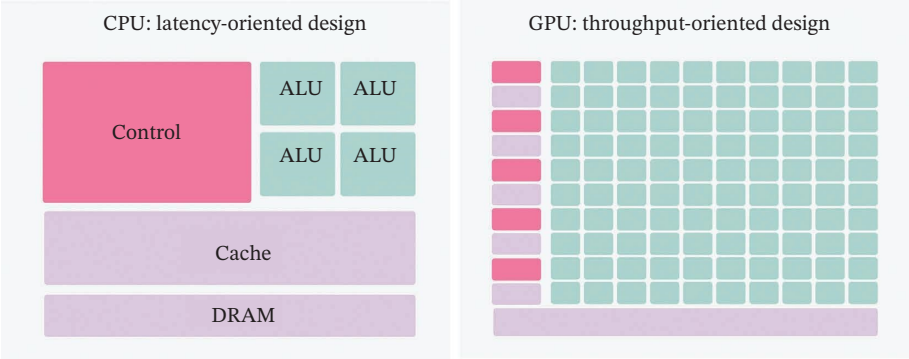

FIGURE 19 | CPU and GPU general architecture.

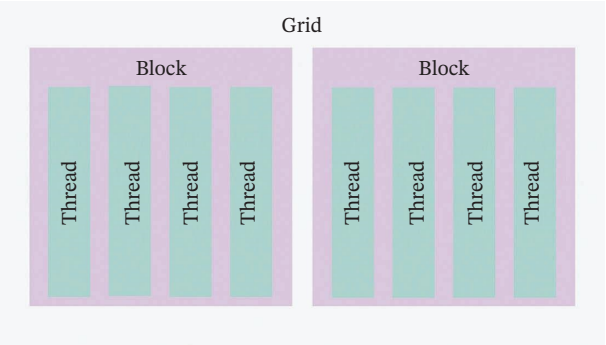

FIGURE 20 | Grid, blocks, and threads representation in GPU.

this aspect is quite crucial. The other reason why this is relevant is because we have a new element in hardware with abilities (data memory management and processing) that were restricted only to CPUs in the past.

A key objective that directed the research is how to apply the parallel mindset to the work. This is not an automatic or trivial step. Imagine that you have a 1000 tasks to process, each with 1000 independent activities. In a first scenario, we have one single worker to complete the processes. The execution would occur serially, and the time to finish would be the time it takes the worker to complete each activity multiplied by the number of activities. Now the capacity is increased with 1000 workers, what would be the best election. Different scenarios are outlined as follows:

- First scenario. Each worker executes one process completely; that is, 1000 workers execute one process with its activities individually.
- Second scenario. Each worker operates one activity regardless of the process until all the activities are completed. Recalling the definition, this execution is possible because the activities are independent.
- Third scenario. A variation of the second scenario. Now the activities are dependent in a process; that is, an activity must finish to start the next one in the same process. Well, even when this process can be executed by the same 1000 workers, the job of each one is not fully separate; then, they must wait until the previous activity is finished.

As we can see, multiple options are possible to execute a process in parallel. In the last scenario, even if you have a bigger capacity, the parallelization of a job must consider the maximization of autonomous and non-dependent activities between threads. Even when there are options to work with this kind of behavior (atomic operations) [69], this can represent an unnecessary overhead and not exploit the full potential of parallel computing and generate delays in the execution.

Therefore, it was so important to decide of the best alternative to make a process parallel, because it depends on multiple factors such as the type of operations, if there is a coupling between elements in a simulation, or if there is a full independence in data.

Particularly for traffic simulator, we identified the next options:

- Simulate each fully parameterized simulation in a different execution process in parallel.
- Simulate each step of the simulation in a different process in parallel.
- Simulate the behavior of each agent, across its lifecycle, impacting changes during the simulation.

Considering the aspects of the model and looking for an independent and more parallelized simulator, we decided to work with the first option.

To understand better the way GPUs work, we researched frameworks and libraries to run parallel processes in GPUs. The answer we found is a just-in-time compiler named Numba. Numba, according with its official documentation [70], is a just-in-time compiler for Python array and numerical functions (an extra reason why a numerical array-oriented Python simulator is a good option) that gives the power to speed up applications with high-performance functions written directly in Python. Numba generates optimized machine code from pure Python using the LLVM compiler infrastructure. With a few simple annotations, array-oriented and math-heavy Python code can be just-in-time optimized.

Numba [71] supports CUDA GPU programming by directly compiling a restricted subset of Python code into CUDA kernels and device functions following the CUDA execution model. Kernels written in Numba have direct access to NumPy arrays. NumPy arrays are transferred between the CPU and the GPU automatically. So, we worked on how to convert the serial simulator to a parallel one. In the simulator, it was decided to process the preparation and initial setup in the CPU and integrate parallel GPU processing in the execution of each scenario of time in the traffic light. Recalling the described in Model section, in the last paper [44], we started exploring performance analysis in the simulator to find which combination of traffic lights in a VTN with a single cross is the best to maximize traffic flow.

We defined three criteria: Source-sink transfer criteria, inner-flow transfer criteria, and mixed-criteria. Well, in that work, we used the simulator in CPU with MATLAB. Now in this new version we are going to process each combination of traffic lights in a thread separately. As we illustrated in Figure 20, it is possible to execute tasks on threads separately. To achieve this, in general, it is necessary to create a process to be executed in GPU in a parallel fashion; this process is named a kernel. In the simulator a kernel in GPU with Numba was created with the decorator @cuda.jit, directly in Python code. This kernel is a function that executes a full simulation of the model, taking as differentiated parameters the combination of traffic lights times. A block diagram of this version is shown in Figure 21.

Because of the support currently available in Numba, not all the mathematical functions are available, so we explored the official documentation to use those functions that are necessary for the simulator.

We have found that some matrix operations were not available; therefore, it was necessary to create them. We use an option called @cuda.jit (device = True) to create functions directly on GPU only callable from another GPU functions, because it does not affect negatively the performance.

Another issue we found is that there is not an integration method available to execute natively in GPU that matches with the simulator; then in the fourth version of the simulator the method of integration is the one provided by the Math library obtained

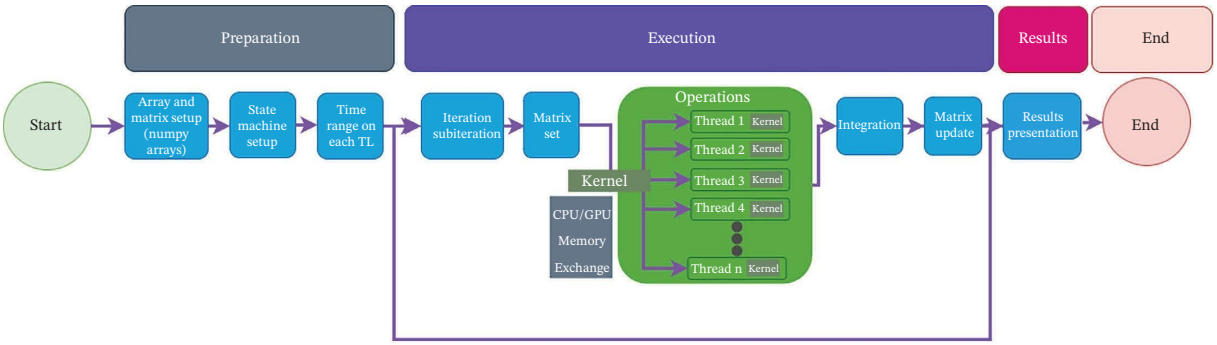

FIGURE 21 | Blocks diagram of the partially GPU version.

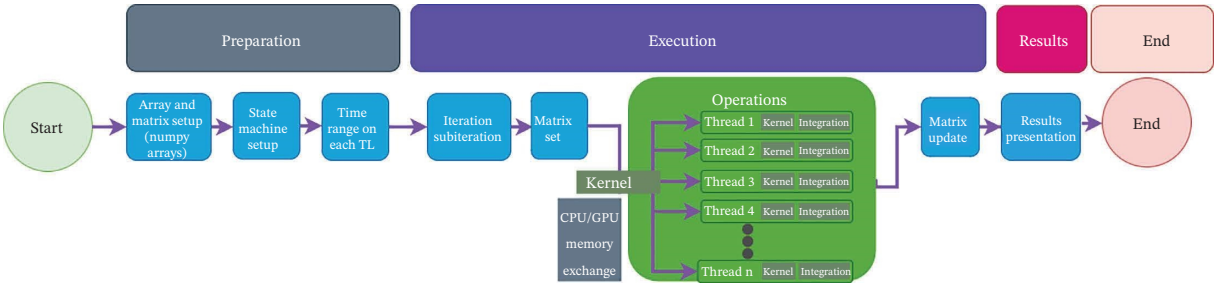

FIGURE 22 | Blocks diagram of full GPU version.

from the CPU. This represents an overhead that impacts negatively performance because the exchange of information between the CPU and GPU is performed in each step of the simulation.

3.4.4 | Fifth Version of Simulator

To avoid the problem of information exchange and following the recommendation of NVIDIA to reduce at minimum the exchange of information between the CPU and GPU, we implemented an integration method directly on GPU that gives the same results as the one existing on CPU to assure the accuracy of the information and comparison. This method was created using the option called @cuda.jit (device = True) to create the function directly on the GPU. Unlike the fourth version, this new option of the simulator performs each cycle of the simulation on the GPU. Only once is the information from the CPU transferred to the GPU with the performance gain. A block diagram of this version is shown in Figure 22.

The algorithm of this version was generated with a diagram showing the algorithm of this version, shown in Algorithm 2, where β is the selected VTN represented as segments of streets, $tl1_{\min}$, $tl1_{\max}$, $tl2_{\min}$, $tl2_{\max}$ depict the parameters of each traffic light, in this case for a single crossing; and finally, z for the number of vehicles initially represented as a scalar in each β, then measured as flow in z_p.

3.4.5 | Results by Version

We conducted different tests in the simulator options that are mentioned above. We ran the same parameters in each version of simulator in a workstation with an Intel Xeon W-2245 3.0“GHz”, 128 GB RAM, GPU NVIDIA Quadro RTX 8000 (Turing) 48 GB RAM, OS Ubuntu. The characteristics of each simulator are presented in Table 1.

ALGORITHM 2 | GPU NumPy algorithm.

Input β, $tl1_{\min}$, $tl1_{\max}$, $tl2_{\min}$, $tl2_{\max}$, z_0

Output z_p

simulations $\leftarrow (tl1_{\max} - tl1_{\min}) * (tl2_{\max} - tl2_{\min})$

blocksPerGrid

threadsPerBlock

Compute z_p parallel in GPU threads

Integrate z_p parallel in GPU threads

Return z_p to CPU

Execution configuration consisted in the definition of multiplication of BlocksPerGrid and ThreadsPerBlock according with the number of scenarios in each simulation (depending the number of seconds of each traffic light to be tested).

We utilized mainly global and thread memory according with hierarchy subsystem [72] of memory from NVIDIA.

We reduced at minimum memory exchange between CPU and GPU, using techniques like passing all the necessary data to GPU once, creating GPU-only accessory functions.

Creating empty arrays from the beginning to be used for the GPU, and be filled during execution and returned only once for CPU to measure results.

In the case of Streaming Multiprocessors (SMs), we considered the rule of having numbers of thread, as a multiple of 32 according with the best practices issued by NVIDIA [72].

TABLE 1 | Different versions of the simulator.

Simulators			
Version	**Language**	**Processing unit**	**Approach**
1	MATLAB	CPU	Serialized
2	Python OOP	CPU	Serialized
3	Python NumPy	CPU	Serialized
4	Python NumPy	CPU and GPU	Parallelized, integrator serialized
5	Python NumPy	CPU and GPU	Fully parallelized

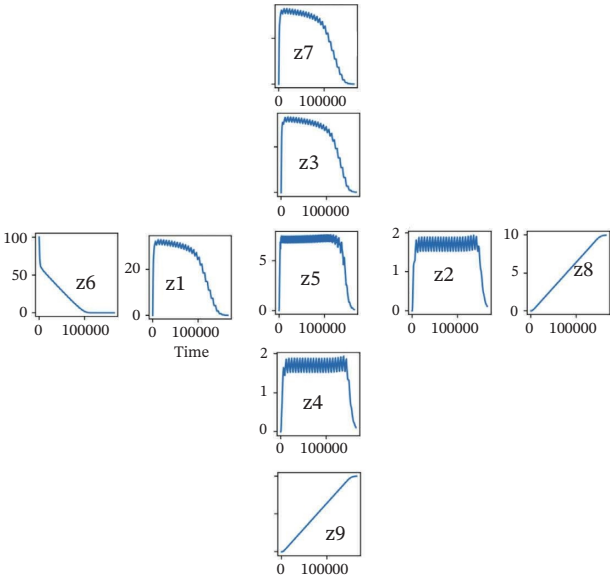

FIGURE 23 | Charts in MATLAB version.

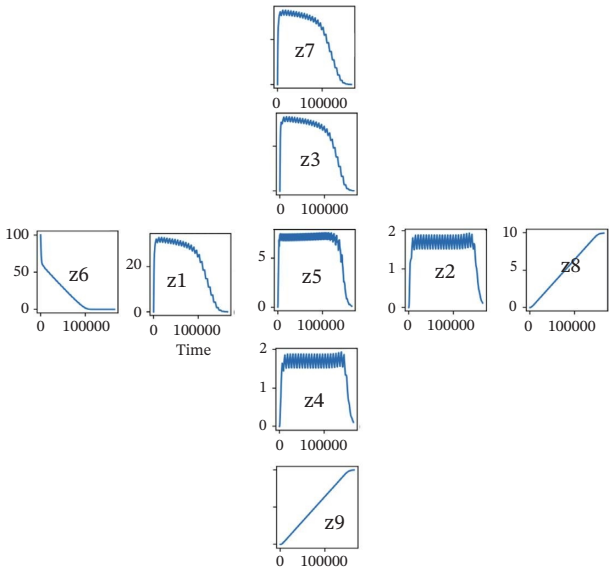

FIGURE 24 | Charts in classes version.

The number of blocks per grid followed the rule defined by NVIDIA according with the specific features by each GPU card (2 x -4x the number of SMs in a card) [72].

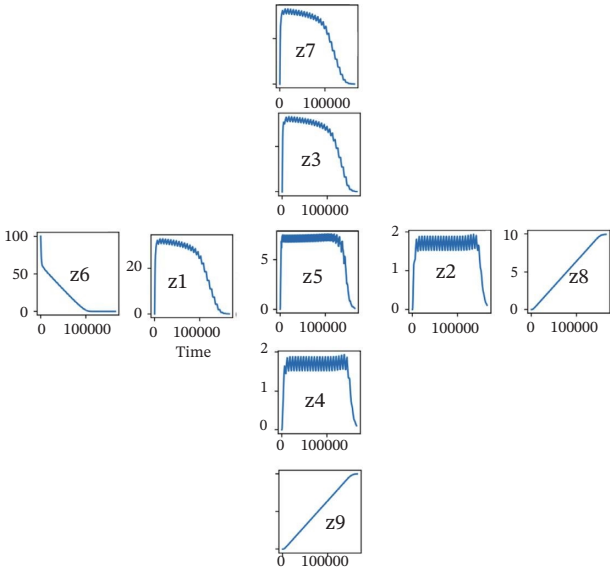

FIGURE 25 | Charts in arrays version.

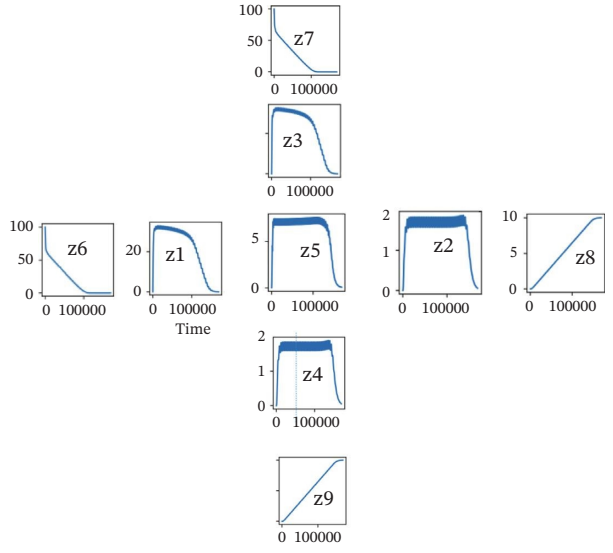

FIGURE 26 | Charts in GPU version (fourth and fifth versions).

Configuring a VTN with a crossing segment that contains two traffic lights, one of them controlling the vertical flow and the other controlling the horizontal flow. A combination of green-time possibilities in each traffic light was defined.

TABLE 2 | Time in minutes by each version.

Results					
Executions (millions)	**MAT-LAB**	**Classes**	**NumPy**	**P-GPU**	**F-GPU**
0.18	0.3	0.9	0.7	7.4	0.4
10.89	8.3	69.8	17.6	4.0	4.1
151.29	35.9	699.7	231.7	11.4	17.6
590.49	254.7	2401.6	819.5	75.0	68.7
1544.490	671.8	3101.3	1051.2	200.0	99.6

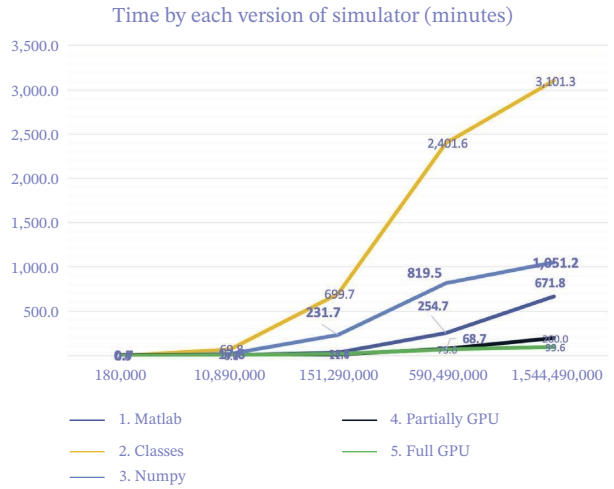

FIGURE 27 | Comparison between versions of the simulator.

The main objective is to measure the time spent by each version to get the results while increasing the possibilities in the VTN. In the simulator were settled different scenarios to measure the best combination that allows a better flow of vehicles. Thinking about the opportunity of having the information in the best period, measurements have been performed to check the time each simulator spent to finish the simulation.

The first step was checking the correctness of the information between simulators and taking as a basis the MATLAB version. We checked the data generated step by step by each version of the simulator to verify whether they had the same results.

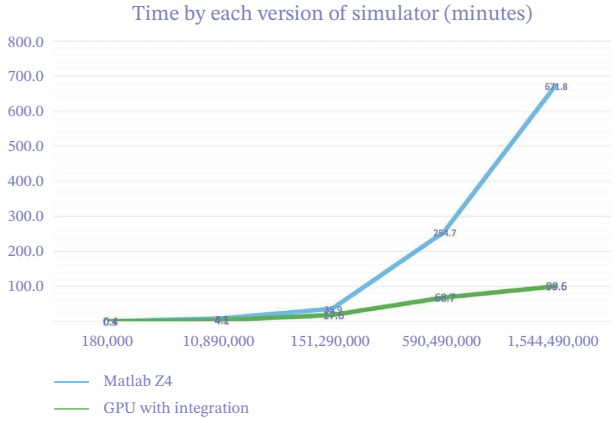

FIGURE 28 | MATLAB vs. GPU full version.

TABLE 3 | MATLAB version vs. full GPU version.

Final comparison		
Executions (millions)	**MATLAB**	**Full GPU**
0.18	0.3	0.4
10.89	8.3	4.1
151.29	35.9	17.6
590.49	254.7	68.7
1544.490	671.8	99.6

Originally, we found some relevant differences due to the codification in a new language. We proved with different methods, especially regarding the simulator method, to get the results closer than the original version. After several tests we obtained similar results between the versions. These tests were conducted first at the data level and then by checking the charts. Figures 23, 24, 25 and 26 show the charts plotting the results of each street segment from z_1 to z_9 obtained in each version of the simulator. Some minor differences can be observed due to the speed of the discharge rate caused by the integrator methods.

Time spent by each simulation is shown numerically in Table 2 and graphically in Figures 27 and 28.

As we can see, the results indicate that the use of OOP (classes) is not a good option since the very first simulations; this is a result of the additional workload necessary to access data in an object structure. Even when in theory the objects are well-suited to model agents, we can see in the results that the time spent by the OOP simulator quickly separates from other versions. On the other hand, having a numerical array-oriented solution was crucial to the simulator because the access and processing of numerical variables like arrays, vectors, and matrix have better performance. The next versions of simulator (NumPy, partially GPU and full GPU) use this kind of approach, accelerating access and calculations that were slow when using objects.

Thinking in parallel, a numerical arrangement resulting in a simpler and more straightforward integration. GPUs work very well with arrays, and the use of these in GPU kernels is well-documented and efficient. In fact, structures like tuples, dictionaries, or objects are not supported by NUMBA in the GPU, so all the information has to be passed in array form. It is similar to what happened in the beginning of programming when a memory management and the care of data types were processes very rigorous and accurate.

There exists a gain in performance comparing MATLAB and NumPy versions (totally simulated in CPU) against GPU versions. Nevertheless, in the case of the partial GPU this benefit is reduced because of the overhead provoked by the exchange of information between them.

CPU and GPU. Still, the NumPy and GPUs versions perform better than MATLAB; the benefit is greater while we get close to a fully parallelized GPU version. We can see that because if the GPU version has the minimum exchange with the CPU, it has the best benefit in performance. If we take the last test (17,161 simulations, 1,544,490,000 executions), comparing MATLAB version against the partially GPU version is 3.4x faster, but in a GPU full version is 6.7 × faster.

Another result is that while the number of simulations is smaller, the gain between the MATLAB version and GPU full version is smaller, or in some cases, like the first test (2 simulations, 180,000 executions), there is no benefit. That is explained by the fact that GPU performs better with simulations greater than thousands of data and processes.

Finally, the main result is found in Table 3, which compares the original version in MATLAB against the full GPU version, where we can observe that the difference between lines grow while more simulations are added to the simulator. In the case of the GPU having a better performance and the time growth does not correspond with the simulations' growth, that is the result of the parallelization execution where the cores in CUDA perform the calculations without growing the necessary time.

4 | Conclusions and Future Work

We can summarize the following conclusions:

- It was possible to translate the model into a Python NumPy version using the features of the language.
- The use of OOP with larger calculations is not a suitable option for the model.
- GPUs can be used to create a Vehicular Traffic Network based on agents and simulate it.
- Using the same computer, the simulation on GPUs has better performance while the number of simulations and executions are increasing.
- It is evident that the CPU version does not represent a good option when the number of simulations as a VTN grows.
- The overhead generated by the exchange of information between the CPU and GPU is not convenient with a reduced number of simulations.
- The lack of necessary and basic functions like matrix operations or integration methods produces an additional amount of time when translating an existing CPU-based simulator.
- NUMBA is a good option to parallelize Python functions.
- In general, the optimization of fetching data from traffic actors (in our case, traffic lights) could serve Transit Departments in Government Institutions to have data more efficiently and test scenarios that best match the needs in each specific location.

In our case, urban planning and signal optimization could be two of multiple tasks where our simulator can help in a process where fast decisions could represent saving minutes in each trip.

Regarding limitations we critically identified the following:

- In the development of the simulator, we identified that the available functions in GPU are limited, moreover talking about NUMBA framework. Considering extra features could result in math and generic functions that would be necessary but not available.
- As it was mentioned before, the hierarchy memory in GPU is limited, then strategies to address this problem would be necessary in bigger scenarios.
- The availability of GPUs both cloud and on-premises when running our approach could be a limitation.
- Even when we started using Python as a starting point due to its relevance to scientific, HPC, and AI cases, we could include a hybrid approach using C++ as the language to enhance some capabilities.
- The use of GPU to have results in different scenarios is one of the possible approaches. Nevertheless, some other options like "attach" an agent to each Core in the GPU could represent a good choice that could be aligned with an agent mindset.

As future work, we are considering the exploration and tests in bigger and complex cases and the integration of quantum computing to optimize the best scenario to improve traffic flow.

Funding

The research and publication of this article was funded by Universidad Iberoamericana A.C. Ciudad de México. The research of this article was funded too by SECIHTI through the scholarship provided to the main author of this article.

Conflicts of Interest

The authors declare no conflicts of interest.

Data Availability Statement

The data that support the findings of this study are available from the corresponding author upon reasonable request.

References

1. Worldometer, "World Population Projections Projections," (2022), https://www.worldometers.info/world-population/.world-population-projections/.

2. World Economic Forum, "Commuters in These Cities Spend More Than 8 Days a Year Stuck in Traffic," https://www.weforum.org/agenda/2019/02/commuters-in-these-cities-spend-more-than-8-days-a-year-stuck-in-traffic/.

3. S. De Silva, A. S. Ball, T. Huynh, and S. M. Reichman, "Metal Accumulation in Roadside Soil in Melbourne, Australia: Effect of Road Age, Traffic Density and Vehicular Speed," *Environmental Pollution* 208 (2016): 102–109, https://doi.org/10.1016/j.envpol.2015.09.032.

4. A. Ali, N. Ayub, M. Shiraz, N. Ullah, A. Gani, and M. A. Qureshi, "Traffic Efficiency Models for Urban Traffic Management Using Mobile Crowd Sensing: A Survey," *Sustainability* 13, no. 23 (2021): 13068, https://doi.org/10.3390/su132313068.

5. A. Pahnabi, K. Ali, and M. Mobara, "Control of Urban Traffic Network Based on Mixed Logical Dynamical Modeling and Constrained Predictive Control Approach," *International Journal of Scientific Engineering and Technology* 5, no. 6 (2016): 367–371.

6. B. Chen and H. H. Cheng, "A Review of the Applications of Agent Technology in Traffic and Transportation Systems," *IEEE Transactions on Intelligent Transportation Systems* 11, no. 2 (June 2010): 485497–497, https://doi.org/10.1109/tits.2010.2048313.

7. S. P. Hoogendoorn and P. H. L. Bovy, "State-of-Theart of Vehicular Traffic Flow Modelling," *Proceedings of the Institution of Mechanical Engineers—Part I: Journal of Systems and Control Engineering* 215, no. 4 (2001): 283–303, https://doi.org/10.1177/095965180121500402.

8. F. van Wageningen-Kessels, H. van Lint, K. Vuik, and S. Hoogendoorn, "Genealogy of Traffic Flow Models," *EURO Journal on Transportation and Logistics* 4, no. 4 (Dec 2015): 445–473, https://doi.org/10.1007/s13676-014-0045-5.

9. T. Bellemans, B. De Schutter, and B. De Moor, "Models for Traffic Control," *Journal A* 43, no. 3 (2002): 13–22.

10. A. M. R. Borsche and A. Meurer, "Microscopic and Macroscopic Models for Coupled Car Traffic and Pedestrian Flow," *Journal of Computational and Applied Mathematics* 348 (2019): 356–382, https://doi.org/10.1016/j.cam.2018.08.037.

11. Yu Yue, A. El Kamel, G. Gong, and F. Li, "Multi-Agent Based Modeling and Simulation of Microscopic Traffic in Virtual Reality System," *Simulation Modelling Practice and Theory* 45 (2014): 62–79, https://doi.org/10.1016/j.simpat.2014.04.001.

12. Y. Kawasaki, Y. Hara, and M. Kuwahara, "Real-Time Monitoring of Dynamic Traffic States by State-Space Model," *Transportation Research Procedia* 21 (2017): 42–55, https://doi.org/10.1016/j.trpro.2017.03.076.

13. D. Ni, "There is Something More Fundamental Than Fundamental Diagram," *Transportation Research Part B: Methodological* 195, no. 103206 (2025): 103206, https://www.sciencedirect.com/science/article/pii/S0191261525000554, https://doi.org/10.1016/j.trb.2025.103206.

14. E. Lee and E. Lee, "Congestion Boundary Approach for Phase Transitions in Traffic Flow," *Transportation Business* 12, no. 1 (2024): 072024, https://doi.org/10.1080/21680566.2024.2379377.

15. Q. Xue, Y. Zhang, X. Yan, and B. Wang, "Analysis of Reaction Time During Car-Following Process Based on Driving Simulation Test," in *2015 International Conference on Transportation Information and Safety (ICTIS)* (Wuhan, China, 2015), 207–212.

16. Y. Yuan, D. Aurélien, and H. van Lint, "Mesoscopic Traffic State Estimation Based on a Variational Formulation of the lwr Model in Lagrangian-Space Coordinates and Kalman Filter," *Transportation Research Procedia* 10, no. 82 92 (2015).

17. Y. Wang, S. Xing, C. Can, et al., "Generative AI for Autonomous Driving: Frontiers and Opportunities," (2025), https://arxiv.org/abs/2505.08854.

18. Y. Gao, M. Piccinini, Y. Zhang, et al., "Foundation Models in Autonomous Driving: A Survey on Scenario Generation and Scenario Analysis," (2025), https://arxiv.org/abs/2506.11526.

19. R. L. Axtell, "120 Million Agents Self-Organize into 6 Million Firms: A Model of the US. Private Sector," in *Proceedings of the 2016 International Conference on Autonomous Agents and Multiagent Systems* (Singapore, Richland, SC, 2016), 806–816.

20. F. Teng, H. Zhang, C. Luo, and Q. Shan, "Delay Tolerant Containment Control for Second-Order Multi-Agent Systems Based on Communication Topology Design," *Neurocomputing* 380 (2019): 11–19, http://www.sciencedirect.com/science/article/pii/S0925231219313736, https://doi.org/10.1016/j.neucom.2019.10.001.

21. Z. Wang and S. Zlatanova, "Multi-Agent Based Path Planning for First Responders Among Moving Obstacles," *Computers, Environment and Urban Systems* 56 (2016): 48–58, http://www.sciencedirect.com/science/article/pii/S0198971515300314, https://doi.org/10.1016/j.compenvurbsys.2015.11.001.

22. D. Liu, S. Baldi, W. Yu, and G. Chen, "On Distributed Implementation of Switch-Based Adaptive Dynamic Programming," *IEEE Transactions on Cybernetics* (2020): 1–7.

23. J. Barceló, *Fundamentals of Traffic Simulation*, 1st ed. (Springer, 2010).

24. K. Małecki and S. Iwan, "Modeling Traffic Flow on Two-Lane Roads With Traffic Lights and Countdown Timer," in *3rd International Conference Green Cities—Green Logistics for Greener Cities, Szczecin*, *Transportation Research Procedia*, 39 (Szczecin, Poland, 2019), 300–308, http://www.sciencedirect.com/science/article/pii/S2352146519301206, https://doi.org/10.1016/j.trpro.2019.06.032.

25. M. Hossain and P.M. McDonald, "Modelling of Traffic Operations in Urban Networks of Developing Countries: A Computer Aided Simulation Approach," *Computers, Environment and Urban Systems* 22, no. 5 (1998): 465–483, http://www.sciencedirect.com/science/article/pii/S0198971598000404, https://doi.org/10.1016/s0198-9715(98)00040-4.

26. Michel dos Santos Soares and J. Vrancken, "A Modular Petri Net to Modeling and Scenario Analysis of a Network of Road Traffic Signals," *Control Engineering Practice* 20, no. 11 (2012): 1183–1194, https://doi.org/10.1016/j.conengprac.2012.06.005.

27. K. M. Ng, M. B. I. Reaz, and M. A. M. Ali, "A Review on the Applications of Petri Nets in Modeling, Analysis, and Control of Urban Traffic," *IEEE Transactions on Intelligent Transportation Systems* 14, no. 2 (June 2013): 858–870, https://doi.org/10.1109/tits.2013.2246153.

28. Z. Wang, Y. Li, and P. Cai, "Supervisory Arc of Petri Nets and Its Application on Traffic Signals Control System," in *International Conference on Automatic Control and Artificial Intelligence (ACAI 2012)* (Xiamen, China, March 2012), 2013–2018.

29. F. Basile, P. Chiacchio, and D. Teta, "A Hybrid Model for Real Time Simulation of Urban Traffic," *Control Engineering Practice* 20, no. 2 (2012): 123–137, https://doi.org/10.1016/j.conengprac.2011.10.002.

30. M. Azzumar, A. Halim, and M. S. Harjono, "Performance Evaluation of Two Ways Urban Traffic Control System Based on Macroscopic Hybrid Petri Net Model," in *2013 International Conference on Advanced Computer Science and Information Systems* (Sept 2013), 335–340.

31. F. Chen, Li Wang, B. Jiang, and C. Wen, "A Novel Hybrid Petri Net Model for Urban Intersection and Its Application in Signal Control Strategy," *Journal of the Franklin Institute* 351, no. 8 (2014): 4357–4380, https://doi.org/10.1016/j.jfranklin.2014.05.002.

32. A. DiFebbraro, D. Giglio, and N. Sacco, "A Deterministic and Stochastic Petri Net Model for Traffic-Responsive Signaling Control in Urban Areas," *IEEE Transactions on Intelligent Transportation Systems* 17, no. 2 (February 2016): 510–524, https://doi.org/10.1109/tits.2015.2478602.

33. L. Qi, M. Zhou, and W. Luan, "Modeling and Control of Urban Road Intersections With Incidents via Timed Petri Nets," in *2015 IEEE 12th International Conference on Networking, Sensing and Control* (Taipei, Taiwan, April 2015), 185–190.

34. M. Janczykowski, W. Turek, M. Malawski, and A. Byrski, "Large-Scale Urban Traffic Simulation With Scala and Highperformance Computing System," *Journal of Computational Science* 35 (2019): 91–101, http://www.sciencedirect.com/science/article/pii/S1877750318306987, https://doi.org/10.1016/j.jocs.2019.06.002.

35. R. L. Axtell, "120 Million Agents Self-Organize Into 6 Million Firms: A Model of the U.S. Private Sector," in *Proceedings of the 2016 International Conference on Autonomous Agents Multiagent Systems, AAMAS '16* (Richland, SC: International Foundation for Autonomous Agents and Multiagent Systems, 2016), 806–816.

36. V. Astarita and V. Pasquale Giofré, "From Traffic Conflict Simulation to Traffic Crash Simulation: Introducing Traffic Safety Indicators Based

on the Explicit Simulation of Potential Driver Errors," *Simulation Modelling Practice and Theory* 94 (2019): 215–236, http://www.sciencedirect.com/science/article/pii/S1569190X19300279, https://doi.org/10.1016/j.simpat.2019.03.003.

37. G. Hou and S. Chen, "Study of Work Zone Traffic Safety Under Adverse Driving Conditions With a Microscopic Traffic Simulation Approach," *Accident Analysis I and Prevention* 145 (2020): 105698, http://www.sciencedirect.com/science/article/pii/S0001457520300877, https://doi.org/10.1016/j.aap.2020.105698.

38. S. C. Calvert and B. van Arem, "A Generic Multilevel Framework for Microscopic Traffic Simulation With Automated Vehicles in Mixed Traffic," *Transportation Research Part C: Emerging Technologies* 110 (2020): 291–311, http://www.sciencedirect.com/science/article/pii/S0968090X19304322, https://doi.org/10.1016/j.trc.2019.11.019.

39. J. Jeong, N. Kim, D. Karbowski, and A. Rousseau, "Implementation of Model Predictive Control Into Closed-Loop Micro-Traffic Simulation for Connected Automated Vehicle," in *9th IFAC Symposium on Advances in Automotive Control AAC 2019*, *IFAC-PapersOnLine*, 52, no. 5 (Orléans, France, 2019), 224–230, http://www.sciencedirect.com/science/article/pii/S2405896319306561, https://doi.org/10.1016/j.ifacol.2019.09.036.

40. I. Overtoom, G. Correia, Y. Huang, and A. Verbraeck, "Assessing the Impacts of Shared Autonomous Vehicles on Congestion and Curb Use: A Traffic Simulation Study in the Hague, Netherlands," *International Journal of Transportation Science and Technology* 9, no. 3 (2020): 195–206, http://www.sciencedirect.com/science/article/pii/S2046043020300265, https://doi.org/10.1016/j.ijtst.2020.03.009.

41. S. C. Calvert, G. Klunder, J. L. L. Steendijk, and M. Snelder, "The Impact and Potential of Cooperative and Automated Driving for Intelligent Traffic Signal Corridors: A Field-Operational-Test and Simulation Experiment," *Case Studies on Transport Policy* 8, no. 3 (2020): 901–919, http://www.sciencedirect.com/science/article/pii/S2213624X20300341, https://doi.org/10.1016/j.cstp.2020.05.011.

42. B. Murat, "Internet of Things-Enabled Unmanned Aerial Vehicles for Real-Time Traffic Mobility Analysis in Smart Cities," *Computers & Electrical Engineering* 123 (2025): 110313, https://www.sciencedirect.com/science/article/pii/S0045790625002563.

43. M. Flores Geronimo, E. G. H. Martinez, E. D. F. Vazquez, J. Job Flores Godoy, and G. F. Anaya, "A Multiagent Systems With Petri Net Approach for Simulation of Urban Traffic Networks," *Computers, Environment and Urban Systems* 89 (2021): 101662, https://doi.org/10.1016/j.compenvurbsys.2021.101662.

44. E. G. Hernandez-Martinez, F. Morales-Torres, J. E. Quiroz-Ibarra, et al., "Modeling and Performance Analysis of Vehicular Traffic Networks: A Multiagent Perspective," in *2022 8th International Conference on Control, Decision and Information Technologies (CoDIT)*, 1 (Istanbul, Turkey, 2022), 797–802.

45. R. A. Horn and C. R. Johnson, *Matrix Analysis* (Cambridge University Press, 1985).

46. Cleve Moler, in *Mathworks 'Scripts'* (2023), https://www.mathworks.com/help/matlab/scripts.html.

47. M. Flores-Geronimo, E. G. Hernandez-Martinez, E. D. Ferreira-Vazquez, J. J. Flores-Godoy, and G. Fernandez-Anaya, "Modular Modelling for Urban Traffic Networks Based on Multi-Agent Systems and Petri Nets," in *2018 XX Congreso Mexicano de Robótica (COMRob)* (September 2018), 1–7.

48. M. Flores-Geronimo, E. G. Hernandez-Martinez, E. D. Ferreira-Vazquez, J. J. Flores-Godoy, and andG. Fernandez-Anaya, "A Hybrid Representation of Urban Traffic Networks Using Multi-Agent Systems and Petri Nets," in *2019 6th International Conference on Control, Decision and Information Technologies (CoDIT)* (Paris, France, April 2019), 1562–1567.

49. Django, "The Web Framework for Perfectionists With Deadlines" (2022).

50. Django, "Overview," in *Software Foundation* (2023), https://www.djangoproject.com/start/overview/.

51. OpenStreetMap, "OpenStreetMap is a Map of the World, Created by People Like You and Free to Use Under an Open License," (2022), https://www.openstreetmap.org.

52. Python Software Foundation, in *CPython Source Code* (2025), https://devguide.python.org/internals/exploring/.

53. Numpy org, in *The Fundamental Package for Scientific Computing With Python* (2022), https://numpy.org.

54. SciPy, in *Integration and Odes* (2024), https://docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.odeint.html.

55. D. B. Kirk and W. mei W. Hwu, "Heterogeneous Parallel Computing," in *Programming Massively Parallel a Hands-on Approach* (Morgan Kaufmann, 2017), 2–10.

56. Anaconda, "Supported Features," (2025), https://numba.readthedocs.io/en/stable/cuda/cooperative_groups.html%23;supportedfeatures.

57. N. I. Dourvas, G. C. Sirakoulis, and A. I. Adamatzky, "Parallel Accelerated Virtual Physarum Lab Based on Cellular Automata Agents," *IEEE Access* 7 (2019): 98306–98318, https://doi.org/10.1109/access.2019.2927815.

58. S. Li, L. Lei, Y. Hu, et al., "A GPU Parallelization Scheme for 3D Agent-Based Simulation of in-Stent Restenosis," in *2019 IEEE International Conference on Cyborg and Bionic Systems* (CBS, 2019), 322–327.

59. A. Rahman, N. A. W. A. Hamid, A. R. Rahiman, and B. Zafar, "Towards Accelerated Agentbased Crowd Simulation for Hajj and Umrah," in *2015 International Symposium on Agents, MultiAgent Systems and Robotics (ISAMSR)* (2015), 6570.

60. S. Xiong, Y. Yao, M. Berry, H. Qi, and Q. Cao, "Frequent Traffic Flow Identification Through Probabilistic Bloom Filter and Its Gpubased Acceleration," *Journal of Network and Computer Applications* 87, no. 60-72 (2017): 60–72, http://www.sciencedirect.com/science/article/pii/S108480451730108X, https://doi.org/10.1016/j.jnca.2017.03.006.

61. S. Bilotta and P. Nesi, "Traffic Flow Reconstruction by Solving Indeterminacy on Traffic Distribution at Junctions," *Future Generation Computer Systems* 114 (2021): 649–660, http://www.sciencedirect.com/science/article/pii/S0167739X20308359, https://doi.org/10.1016/j.future.2020.08.017.

62. E. Egea-Lopez, F. Losilla, J. Pascual-Garcia, and J. M. Molina-Garcia-Pardo, "Vehicular Networks Simulation With Realistic Physics," *IEEE Access* 7 (2019): 44021–44036, https://doi.org/10.1109/access.2019.2908651.

63. M. Edmonds, T. Atahary, S. Douglass, and T. Taha, "Hardware Accelerated Semantic Declarative Memory Systems Through Cuda and Mapreduce," *IEEE Transactions on Parallel and Distributed Systems* 30, no. 3 (2019): 601–614, https://doi.org/10.1109/tpds.2018.2866848.

64. A. Saprykin, N. Chokani, and R. S. Abhari, "GEMSim: A GPU-Accelerated Multimodal Mobility Simulator for Large-Scale Scenarios," *Simulation Modelling Practice and Theory* 94 (2019): 199–214, https://doi.org/10.1016/j.simpat.2019.03.002.

65. A. Saprykin, N. Chokani, and R. S. Abhari, "Gridlock Resolution in a GPU-Accelerated Traffic Queue Model," *Procedia Computer Science* 170 (2020): 681–687, https://doi.org/10.1016/j.procs.2020.03.171.

66. X. Song, Z. Xie, Y. Xu, et al., "Supporting Real-World Network-Oriented Mesoscopic Traffic Simulation on GPU," *Simulation Modelling Practice and Theory* 74 (2017): 46–63, http://www.sciencedirect.com/science/article/pii/S1569190X17300175, https://doi.org/10.1016/j.simpat.2017.02.003.

67. V. A. Vu and G. Tan, "A Framework for Mesoscopic Traffic Simulation in GPU," *IEEE Transactions on Parallel and Distributed Systems* 30, no. 8 (2019): 1691–1703, https://doi.org/10.1109/tpds.2019.2896636.

68. Intel, "CPU vs GPU: What's the Difference," (2023), https://www.intel.com/content/www/us/en/products/docs/processors/cpu-vs-gpu.html.

69. Anaconda, in *Supported Atomic Operations* (2023), https://numba.readthedocs.io/en/stable/cuda/intrinsics.html.

70. Anaconda, "Overview," (2023), https://numba.readthedocs.io/en/stable/user/overview.html.

71. Anaconda, "Numba for CUDA GPUs," (2023), https://numba.readthedocs.io/en/stable/cuda/index.html.

72. Nvidia, in *Memory Hierarchy* (2025), https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html%23;memoryhierarchy.