Large-Scale Parallel Traffic Simulation across Multiple GPUs

(15-618 Parallel Computer Architecture and Programming)

Mingqi Lu Jiaying Li

Carnegie Mellon University

December 2025

Abstract

Summary: We extend a single-GPU microscopic traffic simulator into a two-GPU system on A100 GPUs, introducing graph partitioning, ghost-zone synchronization and optimized vehicle migration. Our final implementation achieves up to **1.41**× faster microsimulation on an 8.8M-trip Bay Area workload.

1 Background

Our project builds upon an existing single-GPU microscopic traffic simulator [2] designed around standard traffic–flow models, including car following, lane changing, and gap acceptance. At its core, the simulator maintains a detailed representation of a road network and simulates each vehicle's state at every timestep. The original system already leveraged parallelism on a single GPU, but its scalability was fundamentally limited by memory capacity and the compute throughput of a single device. This made it a strong candidate for multi-GPU extension.

Key Data Structures

The simulator uses three primary data structures to represent the road network and the vehicles moving through it.

Lane Map The full network is encoded as a large one-dimensional byte array in GPU memory. Each byte corresponds to one meter of road. A value of `255` indicates that the position is unoccupied, while values from `0`–`254` indicate occupancy along with the vehicle's current speed. This

compact representation allows efficient lookup of headway, neighbor detection for lane changing, and collision checks.

Network Metadata Each edge in the network graph stores its number of lanes, its offset into the lane map, and the indices of its upstream and downstream intersections. This metadata enables mapping a vehicle's physical position to memory offsets and identifies the next road segment during routing.

Vehicle State Each vehicle stores its precomputed route, as well as its previous, current, and next edges. During simulation, the vehicle's position on the current edge is updated, and transitions to downstream edges are handled based on available space in the lane map and intersection status.

Together, these structures allow the simulator to compute vehicle dynamics by translating vehicle-edge relationships into byte-level operations on the lane map.

Key Operations

Each GPU thread is responsible for updating one vehicle per timestep. The main operations include:

- Checking departure conditions and activating the vehicle when its release time is reached.
- Reading the lane map to determine available headway and adjusting velocity according to the car-following model.
- Evaluating intersection behavior, potential lane changes, and transitions to downstream edges.
- Writing updated position and speed back into the lane map.

These operations are inherently data-parallel, as most vehicles move independently except when interacting with neighboring vehicles or contending for lane space.

Inputs and Outputs

The simulator consumes a road network (nodes and edges), OD demand(origin and destination of each vehicle). The output consists of per-vehicle routes and metrics such as travel distance, travel time, and average speed after simulation.

Computational Bottlenecks and Parallelization Opportunities

The dominant computation occurs during each timestep, where every active vehicle updates its state. Since these updates follow a common control flow and frequently access local road information, the workload maps naturally to GPU parallelism.

However, scaling beyond a single GPU introduces significant challenges. A single device is constrained both by memory capacity and by the number of vehicles it can simulate in real time. To support larger networks and higher demand scenarios, the simulator must distribute work across multiple GPUs while preserving correctness at partition boundaries.

Workload Breakdown

The full pipeline consists of the following stages:

- CPU-side preprocessing: reading the network, building the graph, routing vehicles, and generating the lane map.
- Allocation and initialization of GPU memory.
- Kernel execution at each timestep, with one thread per vehicle.
- Final aggregation and output on the CPU.

Among these stages, timestep propagation is by far the most expensive and is the primary target for parallel acceleration, especially when the od demand is large.

2 Approach

Programming Model and Target Hardware

Our implementation is written in CUDA and runs on a two-GPU A100 system connected via NVLink. The original simulator treated the entire road network as residing on a single GPU. Extending this model to multiple GPUs required redesigning the data layout, partitioning the graph, and implementing vehicle migration across devices.

Mapping the Computation to the GPU

In the single-GPU design, each vehicle is mapped to one CUDA thread, yielding a uniform SIMD-friendly workload. Vehicle updates follow a largely consistent control flow, with divergence occurring primarily during intersection handling and lane changes.

Transitioning to a multi-GPU design fundamentally changes the data mapping. The road network and associated lane map must be partitioned across devices. Each GPU owns a disjoint subset of intersections and edges, and vehicles are simulated on the GPU corresponding to their current edge.

Edges whose upstream and downstream nodes lie on different GPUs create boundary conditions that require additional coordination. Since we partition the graph by nodes, any edge crossing a partition boundary must be visible to both GPUs during simulation. To ensure correctness, we introduce *ghost zones*, where the entire lane map of a boundary edge is replicated on each adjacent GPU.

The motivation is that vehicles located near the boundary need accurate knowledge of all other vehicles on the same edge, even if those vehicles are physically simulated on another GPU. By duplicating the lane map across different GPUs, both GPUs can safely compute car-following and lane-changing behavior on their local copies without stalling or waiting for remote queries. The vehicle itself remains owned by exactly one GPU.

Modifications to the Original Algorithm

In the single-GPU version, all global state is shared and updates occur through simple writes into a single lane map and vehicle array. In the multi-GPU setting, however, the road network is partitioned by intersections, and edges whose endpoints lie on different GPUs become cross-partition edges. Vehicles traveling on these edges must still observe the full occupancy of the lane map, even though part of that information belongs to another device. To support this, we introduce *ghost edges*, in which the entire lane map of each cross-partition edge is replicated on both GPUs. During simulation, updates to these edges are forwarded between devices to keep their replicas consistent.

Vehicle ownership remains unique even though edge data is replicated. At each timestep,

we track vehicles that move onto edges belonging to another GPU. When a vehicle reaches an intersection and its downstream edge is assigned to a different GPU, the simulation performs a migration in two steps:

- Copy: The vehicle's state is first copied into the destination GPU's data structures so that it can be updated correctly on the next timestep.
- Remove: After the copy, the vehicle is removed from the source GPU, completing the ownership transfer.

This copy-then-remove protocol ensures that there is no timestep in which a migrating vehicle is missing from both devices or appearing on both. Vehicles whose next edge remains within the same GPU follow the same update logic as in the original single-GPU implementation and do not require migration.

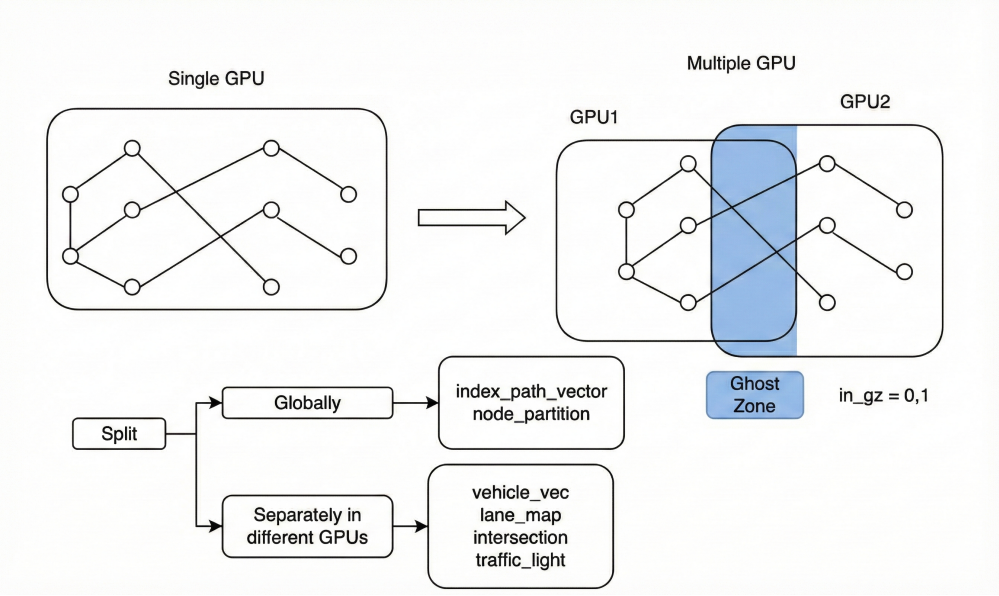

Figure 1: Multi-GPU partitioning using a shared ghost zone to maintain cross-region consistency.

A conceptual overview of this process is shown in Fig.1 and Fig.2 : intersections form the graph nodes, and directed adjacency lists connect them to 1D road-segment arrays. When selecting ghost zones, the entire edge array is replicated on neighboring GPUs so that both devices may safely compute vehicle dynamics simultaneously.

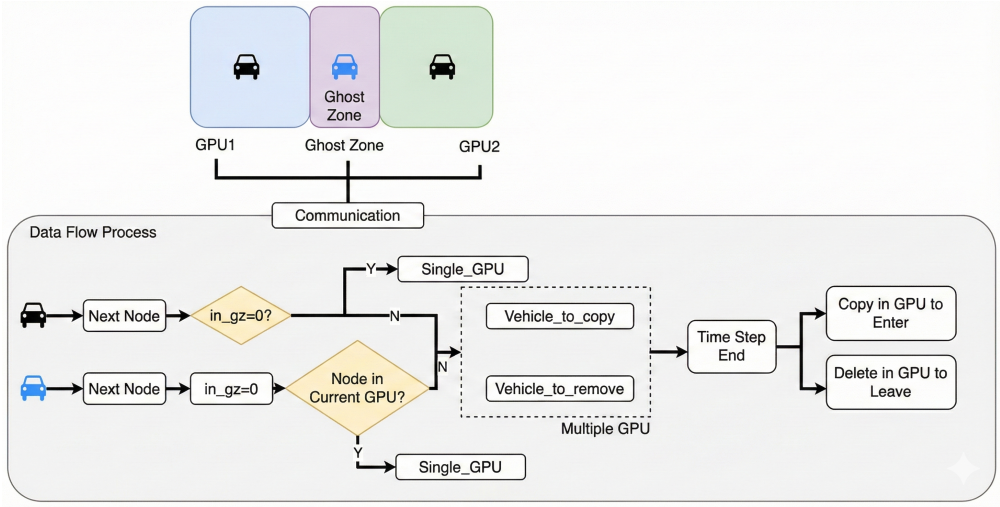

Figure 2: Data flow for vehicle transfer when crossing GPU boundaries.

Graph Partitioning Strategy

Our initial multi-GPU implementation assigned the first half of intersections to GPU 0 and the second half to GPU 1. This index-based split ignored the network's spatial and topological structure, resulting in many major roads being cut across partitions. Vehicles crossed GPU boundaries extremely frequently, and migration plus ghost-edge synchronization quickly overshadowed computation. Even moderate-size networks became impractically slow under this scheme. This made us realize that partitioning, although not part of the core parallel algorithm, is critically important for performance and behaves more like an optimization problem.

To reduce communication, we switched to a community detection [1] partitioning workflow with two stages:

- **Heuristic graph clustering.** We build a weighted adjacency graph from the route data, where edge weights reflect how often intersection pairs co-occur along trips. A METIS-style heuristic produces fine-grained communities with dense internal connectivity and fewer external links.

- **Spatial aggregation.** These communities are then grouped into GPU partitions using k-means clustering on geographic centroids, producing partitions that are both topologically coherent and spatially contiguous.

Although this partitioning is not strictly balanced in node count, it dramatically reduces the

number of cross-partition edges. As a result, vehicle migration becomes rare, boundary synchronization costs decrease substantially, and the simulator operates reliably on full-scale networks.

Inter-GPU Communication

Our multi-GPU design requires frequent vehicle migration and ghost-edge synchronization between devices. During development, we iteratively optimized both the data structures and the communication pattern.

Naive dynamic arrays. In the first prototype, each GPU stored its local vehicles in a raw device array. Because the number of vehicles changes every timestep, we resized this array on every iteration by freeing and re-allocating device memory. This led to:

- repeated `cudaMalloc`/`cudaFree` in the simulation loop, and
- expensive device memory copies when growing or shrinking arrays.

The overhead quickly dominated runtime, and the two-GPU version ran significantly slower than the original single-GPU implementation.

Device-vector based layout. To eliminate the constant `cudaMalloc`/`cudaFree` cycle in our naive array-based implementation. We first considered preallocating a very large static array, but the peak number of vehicles on each GPU is not known in advance and can vary substantially over time; allocating enough space for the worst case would waste memory and still provide no guarantee against overflow on dense networks.

So we switched to `thrust::device_vector<Vehicle>` for storing the per-GPU vehicle lists. Internally, it grows similarly to `std::vector` by doubling its capacity so it can expand as the number of active vehicles changes without requiring manual intervention. Using a device vector gives us several advantages: it automatically handles dynamic resizing, avoids explicit memory bookkeeping, and eliminates the risk of memory leaks caused by manual reallocation. In practice, this simplified the code and stabilized memory behavior.

However, even with device vectors, profiling showed that most of the remaining overhead came from communication operations: copying vehicles across GPUs when they cross partitions, and

removing their old entries on the source GPU. These per-timestep container updates became the dominant cost and limited scalability.

Parallel communication. We implemented an intermediate communication path. Each thread records migration-related events into three device-side buffers:

- the indices of vehicles that should be copied to another GPU,
- the indices of vehicles that should be removed locally after migration, and
- updates to ghost edges (lane-map entries that must be synchronized to neighbors).

After the kernels finish, the host retrieves these cursors and aggregates all per-GPU logs into a single communication plan for the current timestep. This involves grouping vehicle copy requests by (source GPU, destination GPU) pairs, batching deletions per GPU, and collecting all ghost-lane updates by destination region. This aggregation step reduces thousands of small kernel-level events into a small number of well-structured communication tasks.

Communication proceeds in three parts:

- **Ghost-edge updates on the GPU.** Once ghost-lane updates are batched for each GPU, we launch an `updateLaneMap<<<>>>` kernel to apply all modifications in parallel on the device itself. This keeps boundary-edge synchronization entirely on the GPU side and avoids CPU-mediated write loops.
- **Vehicle replication across GPUs.** First, we replaced the original device–host–device path with `cudaMemcpyPeer`, enabling direct GPU–GPU transfers over NVLink. Second, because migrations can occur between multiple GPU pairs in the same timestep, we parallelized these transfers using separate `std::thread` workers (pthread implementations underneath). Each thread is responsible for one source–destination pair, performs its own `cudaSetDevice`, and issues the P2P copy.
- **Vehicle deletions on the source GPU.** Our initial deletion scheme relied on `thrust::remove_if` over the entire container to remove migrated vehicles. But this scans every slot and shifts many elements, which is very expensive.

A key observation is that we do not require vehicles to remain in any particular order, as long as each one keeps an id. We therefore optimized the deletion strategy to:

- collect the indices of vehicles to be removed on each GPU;
- move valid vehicles from the tail into these "holes" using permutation-based copy and scatter operations;
- perform a single resize of the device vector at the end of the timestep.

This turns deletions from an $O(N)$ full scan into work proportional to the number of migrating vehicles, which is typically much smaller than the total population.

Preallocated auxiliary buffers. Both migration and deletion rely on temporary buffers to store indices and intermediate vehicle data. Our initial approach constructed fresh `thrust::device_vector`s inside the simulation loop, which implicitly triggered device allocations and frees every timestep. We eliminated this overhead by moving all buffer allocations to initialization and preallocating sufficiently large auxiliary arrays per GPU. The simulation loop now reuses these buffers without any additional `cudaMalloc`/`cudaFree`, making communication cost scale primarily with the number of migrated vehicles rather than memory bookkeeping.

Replacing blocking threads with asynchronous streams. Before optimization, the communication layer on the host used a pool of worker threads (`std::thread`/pthreads). Each thread issued its own synchronous `cudaMemcpy` call using ordinary pageable host memory. Nsight Systems quickly revealed the downside of this design.

Because the GPU cannot perform DMA directly from pageable memory, every transfer triggers an internal staging step inside the CUDA driver: the data is first copied into an internal pinned buffer before the actual device–host transfer can occur. This extra CPU-side copy happens inside the `cudaMemcpy` call, forcing each worker thread to block until the copy completes. As a result, the main thread also blocks in `pthread_join`, and the GPUs show large idle regions where no kernels can be launched. In practice, this staging effect caused what should have been parallel communication to behave almost entirely serially.

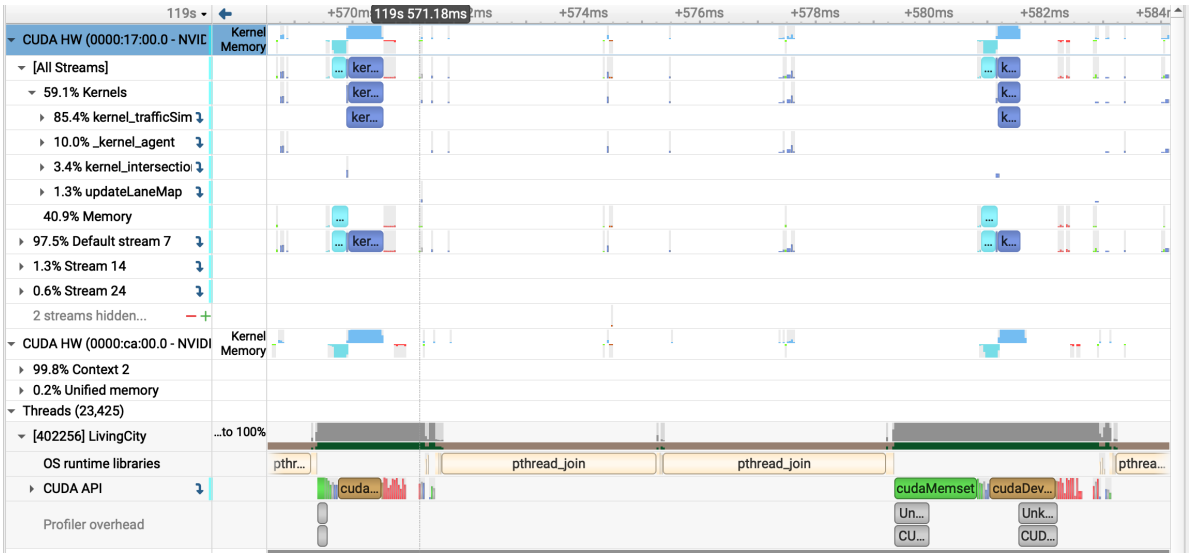

Figure 3: Profiling before optimization. Top: large idle gaps where GPUs cannot issue new work. Bottom: long `pthread_join` segments showing that the CPU is blocked by synchronous `cudaMemcpy` operations on pageable memory.

To remove this bottleneck, we switched all communication buffers to **pinned host memory** using `cudaMallocHost`, and replaced synchronous transfers with **asynchronous** `cudaMemcpyAsync` calls issued on dedicated CUDA streams. Pinned memory allows the GPU to DMA directly from the host buffer, eliminating the hidden staging copy. With `cudaMemcpyAsync`, the CPU now only enqueues the copy operations and immediately continues, rather than waiting for each transfer to finish. This decouples communication from host scheduling and allows transfers to overlap with GPU computation.

The improvement is visible in the optimized profile:

In Fig. 4, GPU streams overlap cleanly, the idle gaps disappear, and the CPU is no longer stalled in `pthread_join`. This confirms that using pinned memory together with asynchronous CUDA streams eliminates the unintended serialization introduced by pageable memory and blocking thread synchronization.

Use of Existing Components

All traffic behavior models such as car following, lane changing and gap acceptance—were inherited from the original single-GPU simulator without modifications. Our work focuses exclusively on enabling multi-GPU execution by introducing partitioning, ghost-edge synchronization, migration

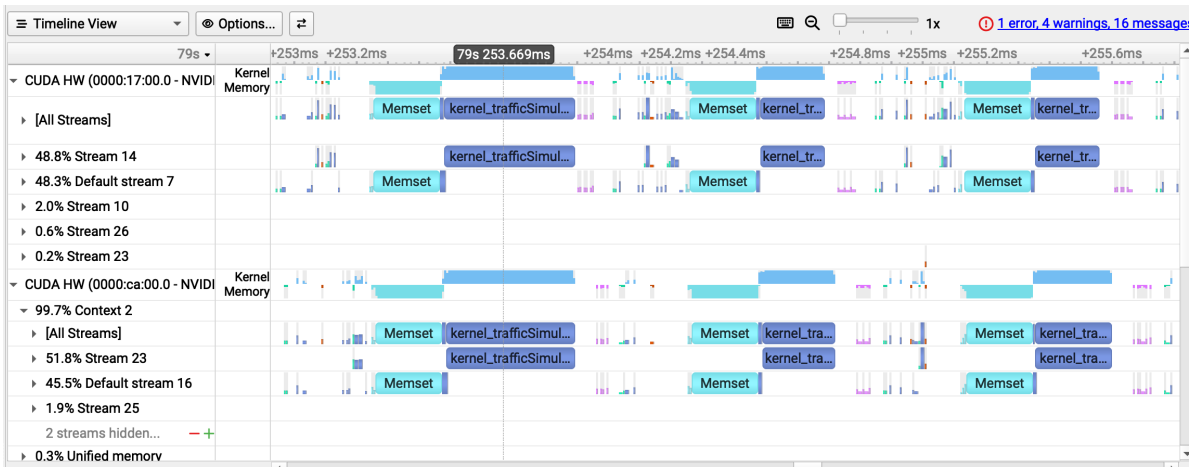

Figure 4: Profiling after optimization.

buffers, and efficient communication mechanisms.

3 Results

3.1 Experimental Setup

We evaluated our optimized multi-GPU traffic simulator on a full-scale, real-world network representing the San Francisco Bay Area.

Network Topology. The road graph contains **223,328 nodes** and **540,827 edges**, extracted from OpenStreetMap. This topology reflects a highly connected urban region with dense intersections and non-uniform road capacities.

Workload Characteristics. The demand dataset spans an entire 24-hour period (00:00–24:00), totaling **8,776,959 trips**. This workload captures realistic temporal variation in traffic density, including morning and evening peaks.

Graph Partitioning. The network was partitioned into two regions using the partitioning method described in Section 2. The resulting distribution assigns approximately **3.7 million** od demands to GPU 0 and **5.1 million** od demands to GPU 1.

Hardware Configuration. All experiments were conducted on a machine equipped with **two A100 GPUs** connected via NVLink.

3.2 Execution Time Breakdown: 1 GPU vs. 2 GPUs

Component	1 GPU	2 GPUs	Difference
Microsimulation time (s)	632.09	447.12	1.41× faster
Total simulation time (s)	817.03	648.74	1.26× faster
Microsim % of total	77.4%	68.9%	—

Table 1: Execution time breakdown for the full 24-hour simulation on 1 GPU and 2 GPUs. Microsimulation includes all per-timestep work: kernels, communication, and migration.

Initially our two-GPU version was actually slower than the one-GPU baseline, because of unoptimized communication. After our iterative optimzation, Table 1 compares the full 24-hour simulation on one GPU and two GPUs. The two-GPU version reduces the microsimulation time from 632 s to 447 s (1.41× faster), which is the dominant component of execution time. This improvement outweighs the modest increase in non-microsimulation due to overhead like partition loading, buffer initialization, pinned-memory setup.

Although the initialization and I/O stages are slightly slower on 2 GPUs, the overall simulation still achieves a 1.26× end-to-end speedup. This confirms that our communication and kernel optimizations successfully accelerate the core timestep loop while keeping synchronization overhead low.

3.3 Scaling Experiment

To evaluate the end-to-end performance of our multi-GPU optimizations, we tested three representative workloads: a light morning peak (6–7 AM), an extended morning period (6–10 AM), and the full 24-hour demand. Each dataset varies significantly in the number of active vehicles, ranging from 0.65 million to 8.77 million agents.

Figure 5 compares the total simulation time on a single GPU versus our optimized two-GPU pipeline. A consistent speedup is observed across all workloads, with larger workloads benefiting more from multi-GPU parallelism. Under light load, communication and synchronization costs are more visible, resulting in a modest 1.07× speedup. In contrast, as the workload grows, the GPUs

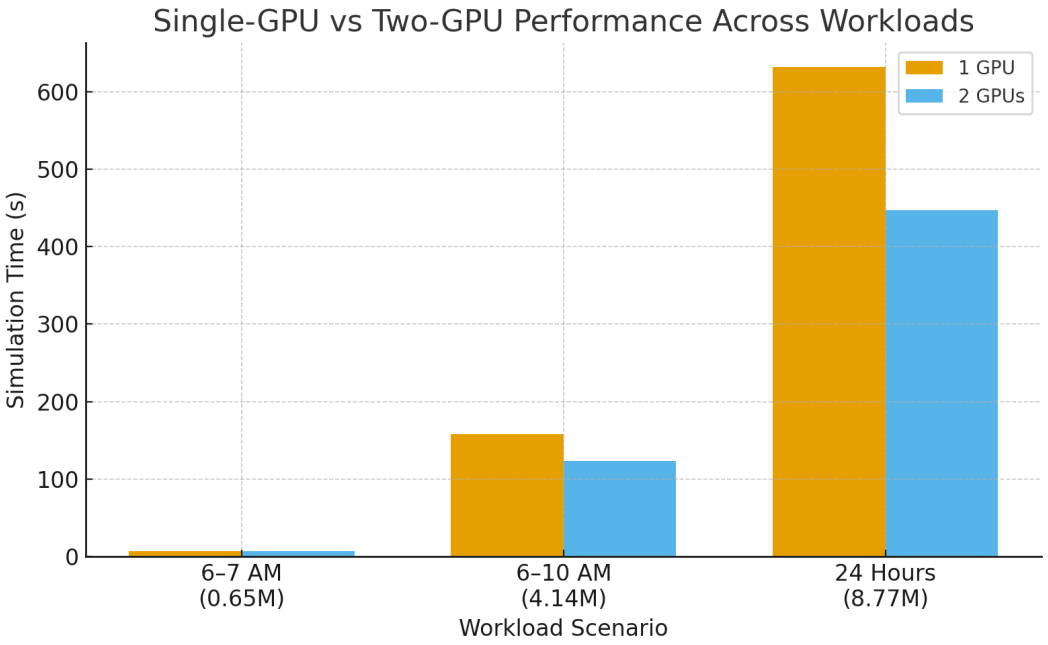

Figure 5: Execution time of the simulator under different workloads. Each scenario compares single-GPU execution against the optimized two-GPU pipeline. Larger workloads exhibit greater speedups due to higher GPU occupancy and improved latency hiding.

become increasingly saturated, allowing our asynchronous communication pipeline to hide memory and transfer latency more effectively. The full 24-hour simulation achieves a speedup of 1.41×.

Table 2 summarizes the quantitative performance improvement.

Scenario	**1 GPU (s)**	**2 GPUs (s)**	**Speedup**
6–7 AM (0.65M vehicles)	7.03	6.55	1.07×
6–10 AM (4.14M vehicles)	157.71	123.10	1.28×
24 Hours (8.77M vehicles)	632.09	447.12	1.41×

Table 2: Performance comparison between single-GPU and two-GPU execution across different traffic workloads.

3.4 Bottleneck Analysis

Although the workload is highly data-parallel, several fundamental constraints prevent ideal 2× scaling on two GPUs:

Load imbalance. After partitioning, GPU 0 processes 3.69M vehicles while GPU 1 handles 5.09M. Because each timestep must finish before the next begins, the slower GPU sets the global

pace. This limits the theoretical maximum speedup to roughly $8.78/5.09 \approx 1.72\times$.

Memory-bound kernel. Each vehicle update performs a realatively small amount of arithmetic operations but issues multiple global memory reads from the road network. These accesses are highly irregular: two adjacent threads often query positions that lie far apart in the 1D lane-map array, and vehicles in the same block rarely operate on the same road segment. Because the lane map is extremely large (on the order of gigabytes) and lacks block-level spatial locality, it might not be tiled into shared memory in meaningful way. Supporting shared-memory caching would require a complete algorithmic redesign, which is far beyond the scope of this project. Consequently, the update kernel remains DRAM-bandwidth-bound.

Synchronous timestep barrier. Even though the per-timestep computation is short, correctness requires that *all* ghost edges and all migrating vehicles be synchronized between GPUs after every timestep. Without this barrier, the two GPUs would have inconsistent views of boundary traffic and the simulation would diverge. This mandatory global synchronization introduces a fixed overhead per timestep that does not shrink with parallelism.

References

[1] Andrea Lancichinetti and Santo Fortunato. Community detection algorithms: A comparative analysis. *Phys. Rev. E*, 80:056117, Nov 2009.

[2] Pavan Yedavalli, Krishna Kumar, and Paul Waddell. Microsimulation analysis for network traffic assignment (manta) at metropolitan-scale for agile transportation planning. *Transportmetrica A: Transport Science*, 18(3):1278–1299, 2022.

Work Distribution

- Jiaying Li(50%) focused on initial multi-GPU implementation and optimizations.
- Mingqi Lu(50%) focused on multi-GPU communication optimizations, graph partitioning and performance profiling.