A Parallel Implementation of Capacitated Vehicle Routing Problem (CVRP) using CUDA

Aida Zametica
Department of Computer Science and Informatics
Faculty of Electrical Engineering
University of Sarajevo
Sarajevo, Bosnia and Herzegovina
azametica1@etf.unsa.ba

Lejla Heleg
Department of Computer Science and Informatics
Faculty of Electrical Engineering
University of Sarajevo
Sarajevo, Bosnia and Herzegovina
lheleg1@etf.unsa.ba

Ajdin Šuta
Department of Computer Science and Informatics
Faculty of Electrical Engineering
University of Sarajevo
Sarajevo, Bosnia and Herzegovina
asuta2@etf.unsa.ba

Selma Ljuhar
Department of Computer Science and Informatics
Faculty of Electrical Engineering
University of Sarajevo
Sarajevo, Bosnia and Herzegovina
sljuhar1@etf.unsa.ba

***Abstract*—The Capacitated Vehicle Routing Problem (CVRP) is a computationally challenging problem that is best know for its real world use cases. Due to its NP-hard nature, the efficient solution of CVRP is critical for applications such as goods delivery, waste collection, and public service logistics. As such, this paper will provide an extension to the classical CVRP by including traffic congestion as an additional constraint while also proposing a parallel solution which leverages genetic algorithms and parallelism on modern-day GPUs. The proposed approach exploits the inherent parallelism of GPUs using CUDA to accelerate the computation process. To evaluate its effectiveness, we perform comparative analyses against existing methods and benchmark the solution across multiple datasets with varying node sizes.**

***Index Terms*—CVRP, CUDA, parallelization, genetic algorithm**

I. Introduction

Vehicle Routing Problems (VRP) are a key challenge in transportation and optimization, with over six decades of research. VRP extends the Traveling Salesman Problem (TSP), requiring the shortest route through a set of cities, visiting each exactly once, and returning to the starting point.

Practical applications of VRP include fleet management, product delivery, and resource distribution. Due to its broad relevance, many VRP variants have been developed, such as the Capacitated Vehicle Routing Problem (CVRP) and the Vehicle Routing Problem with Time Windows (VRPTW). These variations introduce constraints that increase complexity but allow for better customization to specific needs.

This paper focuses on CVRP, which aims to minimize total distance or cost while ensuring vehicle capacity limits are respected and each customer is visited exactly once. CVRP has practical applications in postal service planning, logistics optimization, waste collection, and more. In addition to the capacity constraint, we will add the traffic jam constraint and analyze its impact.

Multiple research papers have been published based on solving the CVRP. Broadly speaking, the solutions provided by these papers can usually be divided into one of two categories, heuristic and non-heuristic solutions. Heuristic methods, also being the most popular choice, perform a relatively limited exploration of the search space and typically produce good quality solutions within modest computing times. For instance, [1] addresses the CVRP using a genetic algorithm implemented on the specialized NVIDIA CUDA architecture, leveraging parallel computation to achieve high computational speeds and near-optimal solutions for smaller benchmark instances. By hybridizing GA with local search algorithms, solution has achieved greater execution times when comparing to sequential versions of the algorithm. Although [1] provides near-optimal solutions for smaller problem instances, its performance deteriorates as the number of delivery customers increases, both in terms of execution time and solution quality. Addressing these limitations is one of the primary objectives of the proposed solution. Other proposed GAs, such as [2] and [4], are based on similar techniques while focusing mainly on improving execution times. For the other type of solutions, nonheuristic, the paper [3] proposes on improving the original *Clarke-Wright Saving Algorithm for CVRP* from 1964 which builds routes by combining nodes that give saving in cost compared with a single route for each node. This method is usually better in terms of time, but may give worse results. [3] Additionally, [5] employs the Upper Confidence Bound for Trees (UCT) to address CVRP with traffic jams, demonstrating significant improvements in time efficiency under complex traffic constraints. A simulation-based approach is proposed in [6], integrating traffic jam constraints and providing pseudocode, which highlights its practical applicability and ability to model real-world scenarios effectively.

II. Problem description and solution

Our exploration into optimizing the CVRP commenced with an examination of existing solutions, particularly focusing on a foundational code by [1]. This code implemented a genetic algorithm (GA) approach to tackle CVRP. The core of the referenced implementation revolves around a genetic algorithm, a stochastic optimization technique inspired by the principles of natural selection and genetics. The GA tackles CVRP by iteratively evolving a population of candidate solutions.

The implementation follows all fundamental steps of a genetic algorithm:

- Parsing input data, including node coordinates, demand values, and vehicle capacity constraints.
- Computing the cost matrix to determine distances between nodes.
- Randomly initializing the population of candidate solutions.
- Refining the population by correcting duplicates, ensuring all nodes are included, and enforcing capacity constraints.
- Selecting parents for crossover using binary tournament selection.
- Applying crossover to generate new candidate solutions.
- Mutating offspring to maintain constraint feasibility and promote diversity.
- Enhancing solutions through 2-opt local search optimization.
- Preserving population diversity by eliminating duplicates and introducing new solutions.
- After a predefined number of generations, the best solution, identified as the one with the lowest cost, is retrieved for result output, along with performance metrics.

A. CUDA-based optimization

The algorithm is implemented entirely on GPU hardware to leverage CUDA's parallel processing capabilities. Implementation is divided into several key components, each optimized for GPU execution. Using the CUDA Python library with Numba and CuPy, the algorithm runs in a CUDA 12.0 environment on Google Collab. This setup enables seamless integration of GPU-accelerated operations with Python's numerical capabilities.

B. Data Handling

The input data for the CVRP is read from a file and stored in a structured format. The data includes the coordinates of the nodes, their demands, and the vehicle capacity. The data is then transferred to the GPU memory using Numba's CUDA API. The data is stored in 2D arrays, where each row represents a node, and the columns represent the node's label, demand, and coordinates.

C. Kernel Setup and Grid Calculation

In CUDA, the execution of kernels is organized into a grid of thread blocks, where each block contains multiple threads. These threads execute kernel functions to perform operations in parallel, significantly accelerating execution compared to a sequential CPU implementation. Each thread identifies its unique portion of the workload using thread and block indices. Grid-stride loops are employed to handle larger datasets, ensuring that the workload is evenly distributed. Synchronization barriers within kernels ensure that all threads within a block complete their tasks before proceeding.

The grid and block dimensions are crucial for maximizing parallelism and ensuring efficient memory access patterns. For each kernel function in the implementation, the grid and block dimensions are calculated based on the size of the input data and the specific requirements of the operation.

1) Fitness Calculation Kernel: The fitness calculation kernel computes the total distance of the route represented by each individual in the population. The grid is set up such that each thread processes one individual. The number of threads per block is typically set to 128 or 256, which is a common choice for balancing parallelism and resource utilization. The grid size is calculated as:

$$\text{grid\_size} = \left\lceil \frac{\text{population\_size}}{\text{threads\_per\_block}} \right\rceil$$

This ensures that all individuals in the population are processed in parallel, with each thread handling one individual.

2) Crossover and Mutation Kernels: The crossover and mutation kernels are designed to maximize parallelism by assigning each individual to a separate thread. The grid and block dimensions are calculated similarly to the fitness calculation kernel. For the crossover kernel, each thread handles one pair of parents and generates two offspring. The mutation operation is also implemented as a CUDA kernel, which randomly swaps two nodes in the route of an individual. Both kernels are designed to minimize thread divergence and maximize parallelism.

3) Two-Opt Optimization Kernel: The two-opt optimization kernel is more complex, as it involves evaluating multiple swaps for each route. The grid is set up such that each thread processes one route, and within each thread, multiple swaps are evaluated. The grid size is calculated as:

$$\text{grid\_size} = \left\lceil \frac{\text{number\_of\_routes}}{\text{threads\_per\_block}} \right\rceil$$

This setup allows for efficient parallel evaluation of swaps, significantly reducing the time required for the two-opt optimization.

D. Memory Management

Memory management is a crucial factor in GPU-accelerated algorithms, as GPUs have a hierarchical memory structure that includes global memory, shared memory, and registers. Efficient use of these memory types is essential for achieving high performance. In the proposed solution, we use global memory to store large data structures such as the cost table and population array, as these cannot fit into the limited shared memory. However, global memory accesses are slower compared to shared memory due to higher latency. To address this, we employ several techniques to optimize memory usage:

1) Coalesced Memory Accesses: We ensure threads within a warp access consecutive memory locations, allowing the GPU to combine multiple memory requests into a single transaction.
2) Efficient Use of Registers: Registers, the fastest memory on the GPU, store temporary variables and intermediate results. For example, during cost calculation node coordinates are stored in registers, reducing global memory accesses and enhancing performance.
3) Minimizing Redundant Memory Accesses: Each thread accesses only the data it needs. For instance, threads calculate fitness values by accessing only relevant parts of the cost table and population array, minimizing redundant accesses.

E. Convergence and Solution Quality

Convergence in genetic algorithms refers to the population evolving toward an optimal or near-optimal solution. In CVRP, this occurs when route quality stabilizes with minimal further improvement. However, premature convergence can trap the algorithm in a local optimum, degrading solution quality.

The proposed genetic algorithm addresses premature convergence through a clone-restricting mechanism that replaces duplicate solutions, enhancing diversity and exploration. Additionally, the optimized kernel configuration ensures comprehensive search space coverage, preventing the algorithm from stagnating in local optima. For example, if the number of threads or blocks is insufficient, some individuals may not be evaluated properly, resulting in a biased population that converges to a local optimum. However, this kernel setup prevents such issues, ensuring thorough exploration and robust solution quality.

To validate the approach, the Gap metric is introduced as a measure of the obtained solutions quality relative to the best-known solution in the literature:

$$\%Gap = \frac{l_o - l^*}{l^*} \times 100 \tag{1}$$

where l_o is the obtained solution cost and l^* is the best-known cost. Lower %Gap values indicate better performance. Proposed approach consistently achieves lower %Gap values than the CPU version, especially for large datasets, due to parallel processing. A similar trend was observed when compared to the reference solution. This will be explained in more detail in the next section.

III. Results Analysis

A. Sequential implementation and limitations

Drawing directly from the reference code, we developed a sequential version to further analyze performance bottlenecks. It became rapidly apparent that sequential execution is prohibitively slow for CVRP. The core reason lies in the inherently parallel nature of genetic operations, where evaluating fitness, selecting candidates, and performing crossovers can be executed concurrently on different processors. Sequentially handling these operations leads to significant delays, especially as the complexity of the problem scales with the number of customers and route permutations.

B. Varying Datasets with Fixed Generations

In this study, the number of generations was set to 20 and 500 for the reference, proposed, and sequential solutions, with a population size of 100. The algorithms were tested on 10 datasets, shown in Table I with varying numbers of nodes and trucks. Each configuration was run with 10 iterations, and the average results were used to ensure reliable and consistent performance metrics.

TABLE I
PROBLEM INSTANCES AND CORRESPONDING DATASETS. n REPRESENTS THE NUMBER OF NODES, AND k NUMBER OF TRUCKS.

Problem Instance	Dataset
P1	P-n16-k8.vrp
P2	P-n40-k5.vrp
P3	A-n80-k10.vrp
P4	M-n151-k12.vrp
P5	B-n31-k5.vrp
P6	P-n55-k15.vrp
P7	E-n76-k14.vrp
P8	P-n21-k2.vrp
P9	B-n45-k5.vrp
P10	P-n70-k10.vrp

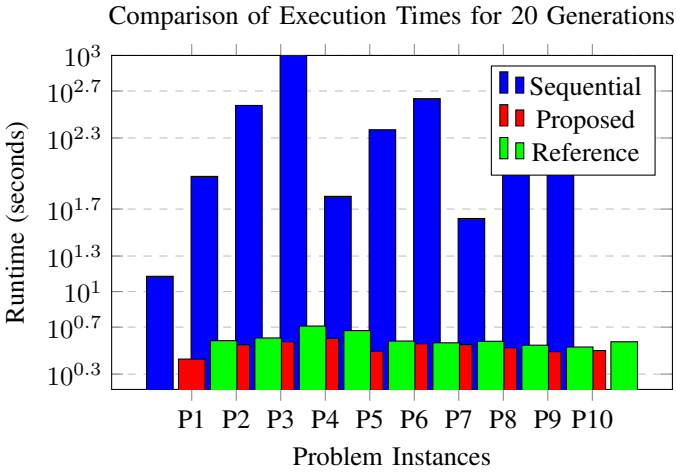

Fig. 1. Execution time comparison for sequential, proposed and reference solution approaches across 10 problem instances tested on 20 generations.

Figure 1 represents execution times for 20 generations. The proposed solution demonstrates significant performance improvements compared to the sequential approach, achieving speedups ranging from ×5.03 to ×342.85 across all problem instances. This highlights the efficiency and scalability of the proposed method, particularly in computationally intensive scenarios. When compared to the reference algorithm, the proposed approach shows marginal improvements, with speedups ranging from ×1.02 to ×1.43, indicating slightly better optimization in execution time.

Only the proposed and reference methods were evaluated for the 500 generation scenario, depicted in Figure 2. The

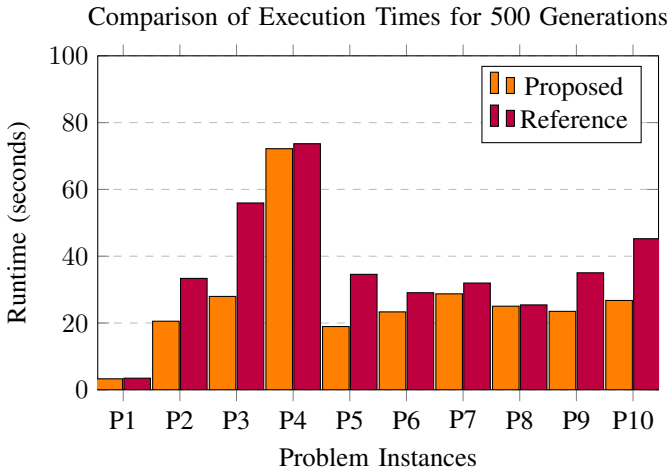

Fig. 2. Execution time comparison for reference and proposed solution approaches across 10 problem instances tested on 500 generations.

proposed method consistently outperformed the reference algorithm, achieving shorter run-times for all problem instances. The performance gap was particularly evident in more complex datasets, such as P3 and P4, highlighting the efficiency of the proposed approach in larger and more demanding scenarios.

C. Varying Generations with Fixed Dataset

In this experiment, the dataset was fixed using the file E-n51-k5.vrp, which contains 51 nodes and 5 trucks, while the number of generations was varied between 500, 1000, 5000, and 10,000. The performance of the proposed and reference algorithms was compared in terms of both the quality of the solution (gap percentage) and the execution time.

The proposed algorithm solution lower gaps in all generations (Figure 3a), demonstrating better convergence with more generations. Unlike the reference approach, whose gap increased for larger numbers (stagnating at higher values), the proposed method consistently delivered more optimal solutions due to improved kernel performance and code cleanup. Notably, for 10,000 generations, the proposed solution's gap is 32.23%, while the reference solution's gap remains at 65.05%, almost double.

One of the key challenges in gap reduction was ensuring proper convergence while avoiding premature stagnation in local minima. Maintaining diversity in solutions was critical, as excessive early convergence could lead to suboptimal results. Mutation played a crucial role in overcoming this issue by preserving exploration capabilities and preventing the algorithm from getting trapped in local optima.

As shown in Figure 3b, the proposed solution required less runtime, particularly at higher generation counts, highlighting its efficiency and suitability for large-scale optimization. For example, at 10,000 generations, the proposed solution takes 390.97 seconds, whereas the reference algorithm takes 744.45

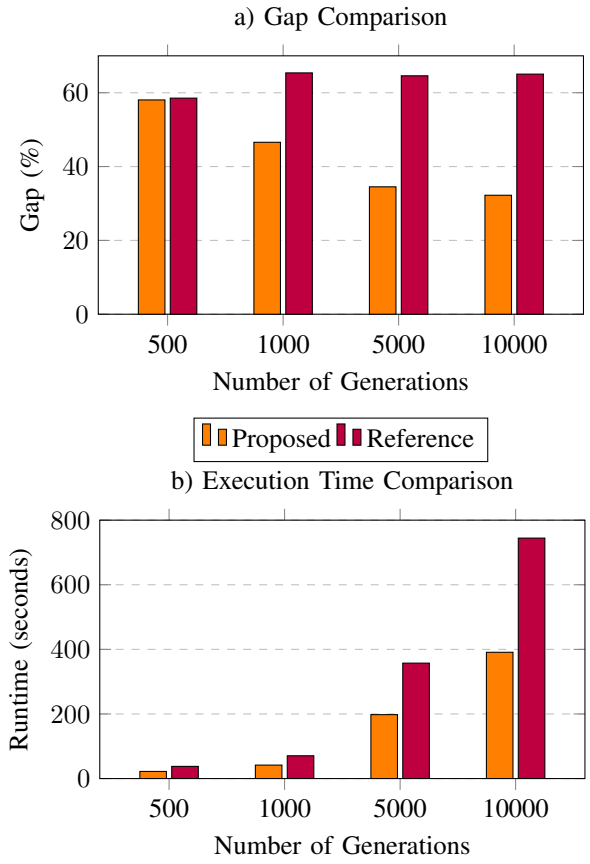

Fig. 3. Comparison of execution time and gap for the reference and proposed solution approaches across different generation sizes on the file E-n51-k5.vrp with 51 nodes and 5 trucks. (a) Gap comparison and (b) Execution time comparison.

seconds, almost twice as much time.

In another experiment, the dataset was fixed using the file P-n40-k5.vrp, which contains 40 nodes and 5 trucks, while the number of generations varied. The performance of the sequential, proposed parallel, and reference parallel algorithms was compared in terms of execution time.

The Figure 4 shows execution time in regards to generation size, with runtime on a logarithmic scale. The sequential approach (blue squares) takes significantly longer than both parallel methods. The proposed parallel algorithm (red triangles) consistently outperforms the reference parallel (green circles), demonstrating better efficiency while maintaining solution quality.

D. CUDA Event Measurement

CUDA event measurement is a technique used to accurately measure the execution time of GPU operations. By recording events before and after kernel execution, it provides precise timing for performance evaluation and optimization. In this study, CUDA event measurement was conducted on the dataset M-n151-k12.vrp, which includes 151 nodes and 12 trucks. The measurement was performed across all functions of the algorithms, with each function executed for 10 iterations, and

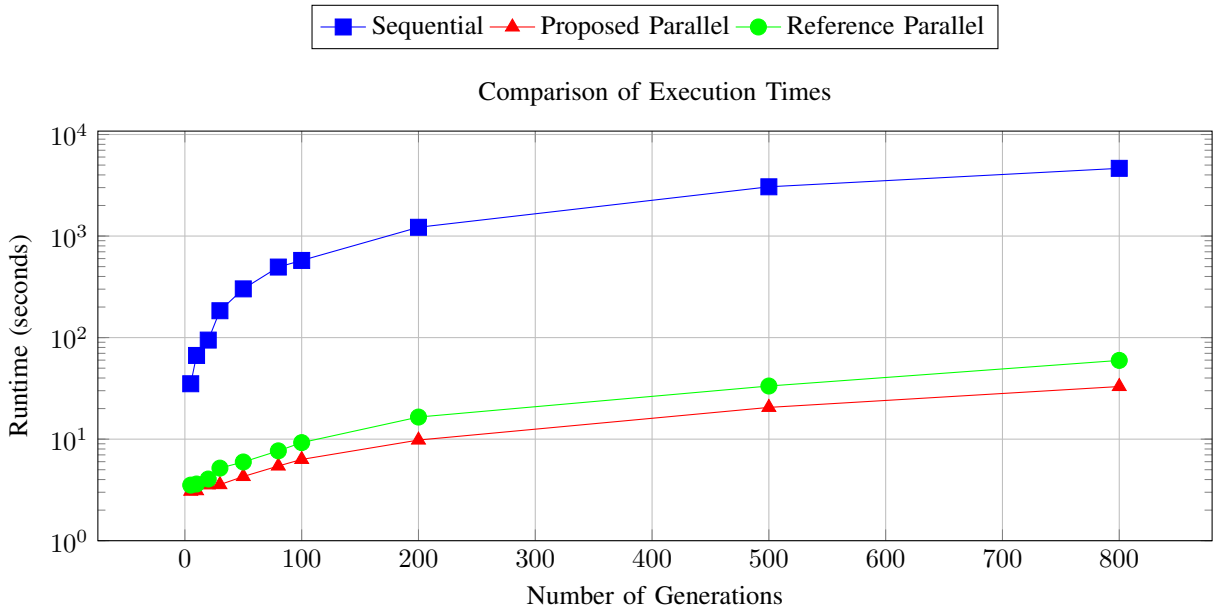

Fig. 4. Execution time comparison for Sequential, Proposed and Reference solutions on the file P-n40-k5.vrp with 40 nodes and 5 trucks.

the average execution time was calculated. The results, shown in Table II demonstrated that the proposed solution consistently outperformed the reference code across all functions, showcasing better execution times and overall efficiency.

TABLE II
CUDA EVENT MEASUREMENTS (IN MILLISECONDS) FOR EACH METHOD.

Function	Proposed solution	Reference solution
calc_cost_gpu	93.1241	334.4353
initializePop_gpu	90.244	175.9992
find_duplicates	202.7673	241.8535
find_missing_nodes	303.7921	230.1167
add_missing_nodes	174.9324	215.8324
shift_r_flag	136.0721	202.8706
cap_adjust	200.1456	340.5959
cleanup_r_flag	142.8137	142.6415
fitness_gpu	283.5497	224.4228
select_candidates	273.5556	295.9316
select_parent	256.5037	373.1482
number_cut_points	340.5583	359.6171
add_cut_points	262.0995	435.3931
cross_over_gpu	346.1803	385.8115
mutate	308.0097	283.429
reset_to_ones	126.1265	169.7472
two_opt	574.6229	592.1755

E. Impact of Traffic Jam Constraint

To evaluate how increasing problem complexity affects the efficiency of the algorithm and the benefits of parallelization, traffic jam constraint was introduced and its impact on execution time was analysed.

Adding the traffic jam constraint required modifying each of the 10 datasets by introducing a section that specifies traffic penalties for each path between two nodes. In the algorithm, this necessitated changes in input processing and the computation of the cost matrix. As a result, when selecting a path, both the distance and the traffic conditions influence the decision. The new optimum was calculated using Google OR-Tools CVRP solver.

The results in Figure 5 and Figure 6 demonstrate the impact of the introduced Traffic Jam constraint and the improvements achieved with the optimized CUDA-based implementation. In both figures, the presence of the traffic jam constraint (red line) increases execution time compared to cases without it (blue line). This confirms that adding this constraint increases the complexity of the problem, leading to longer computation times.

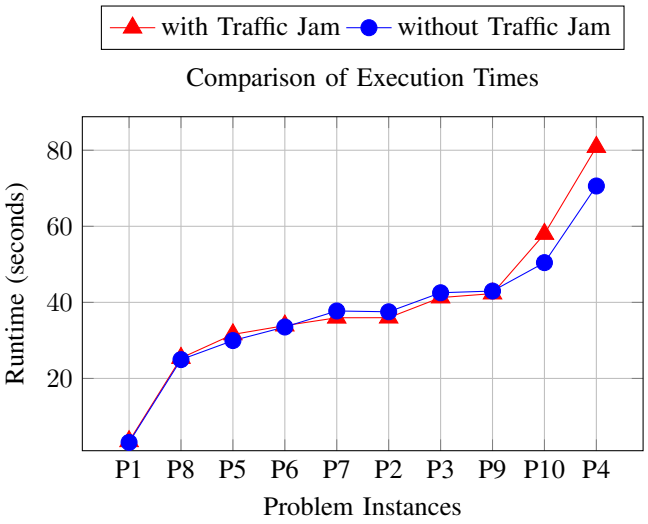

Fig. 5. Execution Time Comparison for Reference Solution

In the reference implementation (Figure 5), the gap between the red and blue lines becomes more pronounced as the

dataset size increases, indicating that the effect of traffic jams intensifies with larger problem instances. In contrast, in the parallel implementation (Figure 6), the two curves remain closer, suggesting that the improved version handles the additional constraint more efficiently and is more resilient to increased complexity.

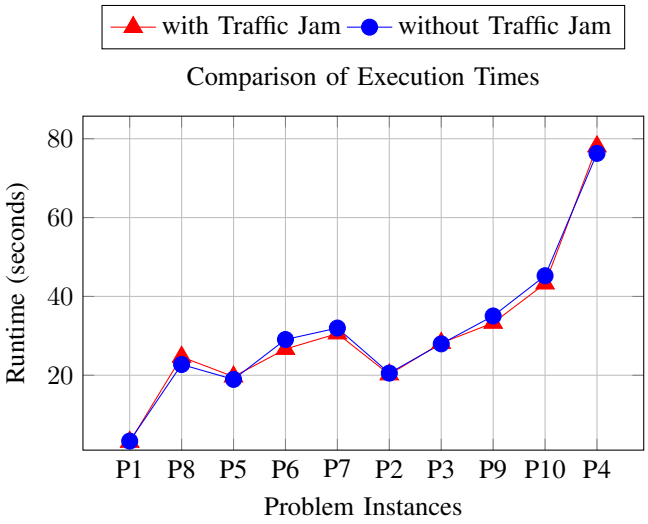

Fig. 6. Execution Time Comparison for Proposed Solution

IV. Conclusion

This paper introduces a CUDA-accelerated genetic algorithm for solving the Capacitated Vehicle Routing Problem (CVRP). By leveraging GPU parallelism, the proposed method achieves significant speedups over sequential and reference implementations, particularly for large datasets. The optimized kernel configurations and code enhancements contribute to improved convergence and solution quality, mitigating premature stagnation in local optima.

Experimental results show that the parallel implementation consistently outperforms both CPU-based and reference GPU methods, achieving up to x342 speedup in execution time while maintaining lower solution gaps. The inclusion of the traffic congestion constraint increases problem complexity but is handled more efficiently in the optimized approach, ensuring better scalability for large-scale transportation and real-world logistics applications.

References

[1] M. F. Abdelatti and M. S. Sodhi, "An improved GPU-accelerated heuristic technique applied to the capacitated vehicle routing problem," 2020.

[2] M. Abbasi, R. Milad, M. R. Khosravi, A. Jolfaei, V. G. Menon, and J. Koushyar, "An efficient parallel genetic algorithm solution for vehicle routing problem in cloud implementation of the intelligent transportation systems," 2020.

[3] A. K. Pamosoaji, P. K. Dewa, and J. V. Krisnanta, "Proposed Modified Clarke-Wright Saving Algorithm for Capacitated Vehicle Routing Problem," 2019. [Online].

[4] P. Yelmewad and B. Talawar, "GPU-based Parallel Heuristics for Capacited Vehicle Routing Problem," 2020.

[5] J. Mańdziuk and M. Świechowski, "UCT in Capacitated Vehicle Routing Problem with Traffic Jams," 2017.

[6] J. Mańdziuk and M. Świechowski, "Simulation-based approach to Vehicle Routing Problem with Traffic Jams," 2016.