A Generic GPU-Accelerated Framework for the Dial-A-Ride Problem

Ramesh Ramasamy Pandi, Song Guang Ho, Sarat Chandra Nagavarapu, *Member, IEEE*, and Justin Dauwels, *Senior Member, IEEE*

***Abstract*— Accelerating the performance of optimization algorithms is crucial for many day-to-day applications. Mobility-on-demand is one such application that is transforming urban mobility by offering reliable and convenient on-demand door-to-door transportation at any time. Dial-a-ride problem (DARP) is an underlying optimization problem in the operational planning of mobility-on-demand systems. The primary objective of DARP is to design routes and schedules to serve passenger transportation requests with high-level user comfort. DARP often arises in dynamic real-world scenarios, where rapid route planning is essential. The traditional CPU-based algorithms are generally too slow to be useful in practice. Since customers expect quick response for their mobility requests, there has been a growing interest in fast solution methods. Therefore, in this paper, we introduce a GPU-based solution methodology for the dial-a-ride problem to produce good solutions in a short time. Specifically, we develop a GPU framework to accelerate time-critical neighborhood exploration of local search operations under the guidance of metaheuristics such as tabu search and variable neighborhood search. Besides, we propose device-oriented optimization strategies to enhance the utilization of a current-generation GPU architecture (Tesla P100). We report speedup achieved by our GPU approach when compared to its classical CPU counterpart, and the effect of each device optimization strategy on computational speedup. Results are based on standard test instances from the literature. Ultimately, the proposed GPU methodology generates better solutions in a short time when compared to the existing sequential approaches.**

***Index Terms*— Dial-a-ride problem, GPU, metaheuristics, tabu search, variable neighborhood search.**

I. Introduction

MOBILITY-ON-DEMAND (MoD) is a transformative and rapidly developing mode of transportation wherein the vehicles transport passengers in a given environment [1]. As autonomous vehicular (AV) technologies continue to emerge, it is more likely that the future MoD systems will rely on AVs to fulfill daily transportation needs. Moreover, MoD has been advocated as a key step toward sustainable urban mobility in the 21st-century [2]. These services often involve transportation of human passengers who place transportation requests with an origin and a destination location. The underlying optimization problem is usually modeled as the dial-a-ride problem, which is an advanced, user-oriented form of mobility service characterized by flexible routing and scheduling. DARP is becoming increasingly relevant in today's public and commercial transit, emergency transportation, ground, and seaport operations.

The vehicle routing problem (VRP) was first introduced in 1959 by Dantzig and Ramser [3]. Dial-a-ride problem (DARP), which is a special class of VRP, arises in the context of demand-responsive transportation. DARP deals with transportation of passengers between pickup and dropoff locations while satisfying constraints such as time-windows, user ride time and route duration limits. DARP is formulated as a combinatorial optimization problem, and the objective is to design vehicle routes and schedules to serve all the users with minimal cost. DARP solution methodologies can be broadly classified into two categories: exact methods and heuristics. Exact methods guarantee optimal solutions while requiring a long execution time, which is suitable for small-scale problems. On the other hand, heuristics can generate solutions quickly without assuring optimality, which is suitable for large-scale problems.

Most of the methods from the literature focus on attaining optimal or near-optimal solutions while little attention has been paid towards developing methodologies to rapidly attain high-quality solutions. In recent times, the advancements in big data and telemetrics have transformed the dial-a-ride (DAR) systems. State-of-the-art technologies yield large amounts of data about transportation networks. The conventional DAR systems require advanced trip booking; however, the planned schedule may need to be adjusted on the fly due to unexpected events. With the advent of advanced machine learning and deep learning approaches, we can estimate the travel times on-the-fly with higher accuracy, and it enables fast algorithms on powerful hardware to achieve "real-time" trip planning. This trend may shift research on fast solution methodologies for DAR systems to facilitate real-time optimization. To achieve the computational speeds necessary, parallel computation (perhaps using metaheuristics) is a promising research direction that is currently lacking in the literature [4], [5].

Graphics processing unit (GPU) technologies have rapidly evolved in the last decade through the advancements in

Manuscript received December 21, 2018; revised June 12, 2019, November 4, 2019, and April 3, 2020; accepted April 21, 2020. This work was supported in part by the ST Engineering-NTU Robotics Corporate Laboratory through the National Research Foundation (NRF) Corporate Lab@University Scheme. The Associate Editor for this article was S.-H. Kong. *(Corresponding author: Ramesh Ramasamy Pandi.)*

Ramesh Ramasamy Pandi, Sarat Chandra Nagavarapu, and Justin Dauwels are with the School of Electrical and Electronics Engineering, Nanyang Technological University, Singapore 639798 (e-mail: ramesh006@e.ntu.edu.sg; sarat@ntu.edu.sg; jdauwels@ntu.edu.sg).

Song Guang Ho is with ST Engineering Land Systems, Singapore 619523 (e-mail: hosongguang@gmail.com).

Digital Object Identifier 10.1109/TITS.2020.2992560

very large scale integration [6]. The application areas of GPUs are as diverse as image processing, machine learning, statistics, 3D reconstruction, game engines, and optimization. GPU technology has the scope of adding even more programmability and parallelism to the ever-evolving hardware architectures. Furthermore, the invention of compute unified device architecture (CUDA) has allowed the users to focus on high-performance computing concepts over the underlying hardware. CUDA programming model that includes C/C++ abstraction has greatly simplified the programmer's burden in building GPU-enabled applications. Although the CUDA abstraction layer hides hardware details from the developer, a comprehensive set of hardware and software level constraints must be satisfied to fully exploit the GPU-acceleration. A CUDA kernel is a set of procedures executed in the GPU, which runs a grid of threads to perform parallel processing in GPU cores. A thread block contains a set of threads segregated into multiple thread warps, where each thread warp contains 32 threads that operate together to execute a single instruction on multiple data at a time. The programmer has control over setting the number of threads and blocks. However, the underlying hardware handles the order of thread execution. The warp scheduler of GPU hardware reduces the data transfer latency by performing arithmetic computation of certain warps while waiting for the read/write process of other warps to complete. Because of this, the arithmetic intensity is kept high at all times to enhance the overall computational efficiency.

Although GPU computing is recognized as an effective way to improve computational efficiency, the development of GPU-based metaheuristics is not straightforward. Various challenges pertaining to hierarchical memory management have to be addressed. Hence, designing GPU-accelerated algorithms based on neighborhood structures for solving real-world optimization problems can be very challenging [7]. To the best of our knowledge, there is no substantive existing work on GPU-based methods for DARP in the literature. This has motivated us to propose a generic GPU-accelerated framework, and integrate it with two well-known metaheuristics: tabu search and variable neighborhood search, for solving DARP. Simulations are conducted on standard benchmark instances from the literature and the results are compared against other relevant methods.

The remainder of the paper is organized as follows. In Section II, we detail the literature review and relevant approaches. In Section III, we describe the DARP formulation. In Section IV, we describe the GPU framework. In Section V, we present the optimization strategies. In Section VI, we detail the GPU-based metaheuristics for DARP. We present the results in Section VII and offer concluding remarks in Section VIII.

II. Literature Review

A. Relevant Studies on DARP

The Dial-a-ride (DAR) system was first introduced by Wilson *et al.* [8] in 1972. Applications have been reported from several cities such as Bologna [9], Copenhagen [10], Milan [11], etc. Several exact methods have been proposed for the single-vehicle [12]–[14], and multi-vehicle [15], [16] cases. The largest problem optimally solved by an exact method has 9 vehicles and 96 requests, that reportedly consumed 899 sec, which in terms of both computational time and problem scale, is far higher to be suited for real-world usability.

To reduce the computation time, heuristics for different versions of DARP were introduced [10], [17]. A heuristic for the transportation of disabled people with a homogeneous fleet of vehicles has been proposed in [18]. A DARP with homogeneous fleet was addressed by a tabu search heuristic in [19] and a variable neighborhood search in [20]. A DARP with transfer points was solved by adaptive large neighborhood search [21]. A DARP with heterogeneous users was addressed by methods such as deterministic annealing [22], evolutionary local search [23], and hybrid genetic algorithm [24]. A DARP with a fixed number of vehicles and whose objective function maximizes service quality for passengers is discussed in [11], [20], [25]–[28]. We study this version of DARP due to its resemblance towards the current DAR services around the world: 1) The constraints of this DARP seems to be in line with the practice of several North American transporters [16]. 2) The information regarding time window widths, vehicle capacity, route duration and maximum ride time was provided by the Montreal Transit Commission (MTC), Canada. The instances are generated according to realistic assumptions [25]. 3) The assumptions of homogeneity of vehicles and their capacities are reasonable in the employment of autonomous electric mini-buses, as nowadays such vehicles mount the same commercial batteries and do not significantly differ in terms of dimensions, capacity, and weights [29]. 4) The assumptions regarding vehicle capacities reflect the situation of transporters using mini-buses for the transportation of individuals or groups of individuals [16]. 5) This version of DARP treats multiple objectives that aim to improve overall service quality which is of primary importance in DARP [25], [30], [31]. 6) Heuristic solutions have been reported in real-time for medium to large-scale problems, which is applicable in the current world since the users appreciate shorter response time [4]. Moreover, the ride-hailing companies have recently started to recognize the importance of response time to customer requests [32]–[34].

In the context of attaining quick solutions (i.e., within 60 sec) for DARP, the Granular Tabu Search proposed by Kirchler and Calvo [31] performs the best when compared to the other existing algorithms, which are (i) Variable Neighborhood Search [20], (ii) Genetic Algorithm [25], and (iii) Tabu Search [19]. We could not find any other algorithms for direct comparison, since many [21]–[23] focus on attaining optimal solution over a long time and do not report solutions in a short time. Therefore, we regard this Granular Tabu Search [31] as the best-performing approach in the context of fast methodologies for DARP. For a broader overview of solution methods and variants of DARP, we refer to [4], a recent survey in the literature.

B. Relevant Studies on GPU-Based Methods

The classical routing and scheduling problems are often solved by local search based heuristics. Such methods usually start from an initial solution which is iteratively improved by exploring the problem-specific neighborhood in the search space. Recently, GPUs are being used for time-efficient neighborhood exploration for different optimization problems.

Several authors have considered the tabu search method as a starting point to integrate a GPU-based local search method to solve the problems not limited to VRP, TSP, and Q3AP. Janiak *et al.* (2008) [35] have solved a traveling salesman problem (TSP) and a flow shop scheduling problem (FSP). TSP is a classical routing problem that aims to design routes for a salesman who must travel between a number of cities. FSP is a classical scheduling problem that aims to design an optimal schedule for a set of jobs to be processed on a set of machines. These authors apply the tabu search method while considering the insertion and swap type of moves in GPU for generating neighborhood solutions. Van Luong *et al.* (2010) [36] have solved a quadratic 3-dimensional assignment problem (Q3AP), which consists of determining permutation matrices to minimize a quadratic function, and has found applications in wireless communication systems [37]. These authors apply swap type of moves during permutations to generate neighborhood solutions in GPU. Czapiński and Barnes (2011) [38] have solved an FSP with a swap-type neighborhood move. When compared to Janiak *et al.* (2008), these authors have evaluated various configurations of the number of threads per block to find the best configuration. Szymon and Dominik (2013) [39] have applied a GPU-based local search with a tabu list to optimize routes for a general vehicle routing problem. Coelho *et al.* (2016) [40] have applied a variable neighborhood search for single-vehicle VRP with pickup and delivery, in which neighborhood solutions are evaluated using GPU.

In comparison with the literature, we solve a different problem known as DARP under the category of VRP. DARP is a difficult problem with time windows and maximum user ride-time constraints, and it generalizes the VRP with pick up and delivery. Attaining a feasible solution for DARP is itself NP-hard since it also generalizes the traveling salesmen problem (TSP) with time windows [16]. The solution space and the evaluation function are problem-dependent. The DARP consists of designing optimal routes and schedules for multiple vehicles to perform door-to-door passenger transit service. We perform insertion-type neighborhood moves similar to Janiak *et al.* [35], however, in our case, we additionally consider pair-precedence constraints, i.e., a drop off node should always be placed after its corresponding pickup node of the request. The kernel configuration is an important factor to maximize the GPU hardware utilization. Since the exhaustive search of the best configuration of threads and blocks for various instances can be time-consuming (as demonstrated by Szymona and Dominik (2013) [39]), we present a method that optimizes the values of the block and thread for the GPU kernels. We develop an evaluation function (see Algorithm 2) to find the feasibility of the DARP solution in GPU. Furthermore, to maximize the GPU utilization, we perform a problem-dependent device-hierarchical memory optimization of data structure, which is regarded as challenging depending upon the considered problem by Van Luong *et al.* (2010) [36]. We have also adapted the branch refactoring method of Imen Chakroun *et al.* (2013) [41] in order to reduce the well-known thread divergence problem of GPU. We perform overlapping data transfers and design multiple kernels deliberately to perform concurrent kernel execution for increased GPU usage. Finally, we also demonstrate the computational speedup achieved by the proposed GPU approach that consists of device optimization strategies in comparison with the conventional approach.

In the literature, ParadisEO [42] is a software framework dedicated to the reusable design of parallel hybrid metaheuristics, in which "the user does not need to build his or her own CUDA code for the specific problem to be solved". On the other hand, our GPU framework has DARP-specific CUDA codes built into it. Our work differs from ParadisEO in the following ways: 1) CUDA kernels that employ concurrent kernel launching scheme, i.e., the kernels are launched simultaneously in an attempt to maximize the computational resource usage. 2) DARP-specific scheme for memory management, i.e., we perform selective data transfer and contiguous memory copy after the kernel executions in order to minimize the CPU-GPU communication latency. 3) Device-specific optimization strategies for the current generation GPU architecture to effectively accelerate DARP solvers, e.g., adaptive kernel grid configuration to dynamically optimize the thread and block size based on the problem size. In summary, the proposed GPU framework is tailored for DARP and has the potential to be integrated with the existing ParadisEO framework. However, we could not make a direct comparison since no results of using this framework have been found in the literature for DARP or any extensions of VRP.

C. Research Gap

There is a growing interest in fast methods since users expect the service providers to provide a solution relatively quickly [4]. This is also evident from the recent dial-a-ride surveys, which suggest that the users tend to prioritize short response time, and the ride-hailing companies have started to act accordingly [32]–[34].

In the context of attaining high-quality solutions in a short time, especially for problems of a realistic scale, the traditional CPU based approaches may be too slow to be applicable. To the best of our knowledge, GPU-accelerated methodologies for DARP are not reported in the literature as stated in Section II-A. Besides, we observe that none of the works in Section II-B show a thorough investigation of fully utilizing GPU (i.e., kernel concurrency, hierarchical device memory optimization, optimizing CPU-GPU communication, reducing thread divergence, adaptive kernel grid configuration). From the scientific perspective, it is critical to design methods that yield better and more robust solutions for important real-world problems such as DARP by fully utilizing the state-of-the-art computing technology.

D. Our Contributions

The main contribution of this paper is the development of a GPU-based solution methodology for DARP with the objective of producing good solutions in a short time. The idea of accelerating local search operation in GPU has been introduced earlier for other optimization problems [35], [36], [38], [39], but never applied for DARP. In our paper, besides applying this conventional GPU approach for DARP, we develop five device-oriented optimization strategies to enhance the utilization of a state-of-the-art GPU architecture.

Specifically, we propose the following device-oriented optimization strategies: i) a device hierarchical memory optimization strategy to effectively map the DARP data structure to various GPU memories, ii) a method to adaptively select the kernel grid configuration for DARP kernels that maximizes GPU utilization, iii) an optimization scheme to reduce data transfer latency, i.e., allocation of data in page-locked memory, segregation of the developed DARP kernels, and overlapping data transfer, iv) a scheme to reduce branch divergence during the process of generating neighborhood solutions for DARP, and v) a concurrent kernel execution for overlapping data transfers between the GPU kernels and CPU functions.

The role of the GPU in the proposed framework is to simultaneously generate and evaluate various solutions during each iteration. The tasks of generating and evaluating such combinations of solutions are very time-intensive, therefore we use GPU for parallel processing. Our GPU approach is generic in the sense that we iteratively select the best admissible solution in CPU followed by GPU-based local search. Here, the choice of requests will be given as input and the evaluated neighborhood solutions will be the output. Therefore, any type of removal and re-insertion operators could be utilized to dislodge the requests involved during the local search operation. Moreover, any metaheuristic that utilizes local search operation could be integrated with the proposed approach. In this paper, we integrate the proposed GPU framework with two metaheuristics: tabu search and variable neighborhood search. The choice of metaheuristics can be justified by their simplicity, efficiency, and applicability for a wide range of problems [43]–[45].

Experiments are conducted on a current-generation GPU, the Tesla P100. Results are based on standard benchmark instances from the literature. Firstly, we demonstrate the speedup achieved by the proposed GPU approach in comparison with its conventional CPU counterpart. We also show the influence of the proposed device optimization strategies on computational speedup. Secondly, we compare the solution quality of our approach with the existing best-performing approaches. Finally, we discuss the effectiveness of our proposed GPU method for DARP under dynamic real-time settings.

The preliminary work of this paper can be found in [46], in which we have described a basic implementation of tabu search on GPU to solve the dial-a-ride problem. In this current version, we have added the following improvements: i) new device-oriented optimization strategies to improve the computational speedup, with extended demonstrations on the effectiveness of each strategy, ii) generalized the GPU framework and integrated with metaheuristics such as tabu search and variable neighborhood search heuristic, and finally compared the solution quality with existing approaches.

TABLE I
TERMINOLOGY

Notation	Description
Depot	Initial and terminal node for each vehicle.
P	Location at which the passenger is picked up.
D	Location at which the passenger is dropped-off.
N	Set of all pick-up points, drop-off points and depot.
n	Total number of customers.
K	Fleet size or a set of vehicles serving the customer requests.
k	A vehicle from the fleet of size m.
E	Set of paths joining the nodes.
T'_k	Route duration of k^{th} vehicle.
T_k	Maximum route duration limit of k^{th} vehicle.
A_i^k	Time at which a vehicle k arrives at node i.
B_i^k	Time at which a vehicle k begins its service at node i.
C_i^k	Time at which a vehicle k ends its service at node i.
D_i^k	Time at which a vehicle k departs from node i.
W_i^k	Waiting time of a vehicle k at node i.
L_i^k	Ride time of the customer i in the vehicle k.
d_i	Service time of the vehicle at node i.
L	Maximum ride time limit of a passenger.
$c(x)$	Total distance traveled by all the vehicles to serve the customers.
Q_i^k	Total number of passengers on the vehicle k after visiting node i.
q_i	Load available at the i^{th} node.
S_k	Seating capacity of the k^{th} vehicle.
e_i	Earliest time at which the service can begin in the node i.
l_i	Latest time beyond which the node i cannot be serviced.
t_{ij}	Travel time between the node i and node j.
t^k	Current time of vehicle k.
x	A DARP solution that consists of m vehicle routes and schedules.
$f(x)$	Objective function of a DARP solution x.
x_{ij}^k	Binary decision variable; 1 if arc (i, j) is traversed by vehicle k and 0 otherwise.

III. Problem Description

The terminology used in this paper is presented in Table I.

In DARP, a fleet of m vehicles represented by a set K has to serve n number of customer requests. The customer requests consist of n pickup and n drop off nodes. The nodes form a directed graph (N, A) graph, where $N = P \cup D \cup \{0, 2n+1\}$, $P = \{1, \ldots, n\}$, and $D = \{n+1, \ldots, 2n\}$. Nodes 0 and $2n+1$ represent the origin and destination depots. Any arc (i, j) denotes a path from the i^{th} node to the j^{th} node. The time taken to travel the arc (i, j) is t_{ij} and the cost of routing is c_{ij}. We adopt the dial-a-ride problem description of [25], which is formulated as follows,

$$\min f(x) = w_1.c(x) + w_2.r(x) + w_3.l(x) + w_4.g(x) + w_5.e(x) + \alpha.k(x), \quad (1)$$

$$\text{subject to,} \sum_{j \in N} x_{0j}^k = 1 \quad \forall k \in K, \quad (2)$$

$$\sum_{i \in D} x_{i,2n+1}^k = 1 \quad \forall k \in K, \quad (3)$$

$$\sum_{j \in N} x_{ji}^k - \sum_{j \in N} x_{ij}^k = 0 \quad \forall i \in P \cup D, k \in K, \quad (4)$$

$$\sum_{k \in K} \sum_{j \in N} x_{ij}^k = 1 \quad \forall i \in P \cup D, \quad (5)$$

$$\sum_{j \in N} x_{ij}^k - \sum_{j \in N} x_{n+i,j}^k = 0 \quad \forall i \in P, k \in K, \quad (6)$$

$$Q_j^k \geq (Q_i^k + q_j) x_{ij}^k \quad \forall i \in N,\ \forall j \in N,\ \forall k \in K, \quad (7)$$

$$\max(0, q_i) \leq Q_i^k \leq \min(S_k, S_k + q_i) \quad \forall i \in N, \forall k \in K \quad (8)$$

$$B_j^k \geq (B_i^k + d_i + t_{ij}) x_{ij}^k \quad \forall i \in N, \forall j \in N, \forall k \in K, \quad (9)$$

$$e_i \leq B_i^k \leq l_i \quad \forall i \in N,\ \forall k \in K, \quad (10)$$

$$L_i^k = (B_{n+i}^k - (B_i^k + d_i))x_{ij}^k \quad \forall i \in P,\ \forall k \in K, \tag{11}$$

$$t_{i,n+i} \leq L_i^k \leq L \quad \forall i \in P,\ \forall k \in K, \tag{12}$$

$$B_{2n+1}^k - B_0^k \leq T_k \quad \forall k \in K. \tag{13}$$

The optimization problem (1) minimizes routing cost $c(x) = \sum_{(i,j)\in A,k\in m} x_{ij}^k t_{ij}$, excess ride time $r(x) = \sum_{i\in n}(B_{i^-} - D_{i^+} - t_{i^+i^-})$, waiting time of passengers on board $l(x) = \sum_{i\in n} w_i(Q_i - q_i)$, route durations $g(x) = \sum_{k\in m} A_{2n+1}^k - D_0^k$, early arrival times at pick up and delivery nodes $e(x) = \sum_{i\in n}(e_i - A_i)^+$, and finally the number of unserved requests $k(x) = n - \sum_{k\in m,ij\in n} x_{ij}^k$. The parameters w_1,w_2,w_3,w_4,w_5,α are statically set according to the application requirement. This objective function has been first introduced in [25].

The constraints specified in Eqs. (2)–(3) ensure that all the vehicle routes start and end at the depot. Eqs. (4)–(5) make it mandatory for every node to be visited by only one vehicle such that any vehicle reaching a particular node must exit it. Eq. (6) ensures that the customers are picked up and dropped off by the same vehicle. Additionally, every vehicle must not exceed its maximum capacity, and the maximum ride time constraints must hold for the vehicles and the customers. Eq. (7) updates the load of the k^{th} vehicle once it reaches node j from node i. Constraint in Eq. (8) ensures that the capacity of vehicle k is not exceeded at any of the nodes. The time consistency is maintained by Eq. (9), which ensures that node j can be serviced only after the arrival of vehicle. Eq. (10) specifies the time window of the service at each location and makes sure that service begins only within the windows. The constraints in Eqs. (11)–(12) guarantee that the maximum ride time of a customer is not exceeded. Finally, the constraint on the route duration of any vehicle k is ensured by Eq. (13).

IV. Proposed GPU Methodology

In this section, we introduce the building blocks of the GPU framework and explain how they are integrated to solve DARP. We observe the following points from our literature review on solving DARP: a) metaheuristics are preferred over exact methods due to their efficiency in attaining good solutions, despite the lack of assurance to achieve optimality, b) most metaheuristics are built upon a local search process. Therefore, we describe the general framework to solve DARP as follows: 1) for a given problem, construct an initial set of vehicle routes, 2) perform a local search process, 3) under the guidance of metaheuristic, choose the next best solution in the defined neighborhood, 4) repeat steps 2-3, until the stopping criterion is satisfied. From our preliminary results, it was identified that the local search process consumes most of the execution time during the program. Therefore, we focus mainly on designing CUDA kernels that perform low-latency local search operation in GPU to efficiently solve DARP.

A. Class Structure

We first create a class that comprises data elements and methods. These methods are device-specific, i.e., they perform operations on the data elements in GPU. We implement the vehicle route as a pointer-to-array, while vertices, route size, and vehicle ID are declared as data elements. Algorithm 1 describes the GPU class. An object of this class can be instantiated to represent a single-vehicle route in GPU.

Algorithm 1 Struct GPU

```
1: int $id$
2: int $start, gap, request, target$
3: int $*route, routesize, routecost$
4: __device__ void dev_scheduler($dev\_problem$)
5: {
6:     Load constraint evaluation
7:     Time window constraint evaluation
8:     Route duration constraint evaluation
9:     Ride time constraint evaluation
10: }
```

To evaluate a vehicle route, we design a GPU-based scheduling heuristic as described in Algorithm 2. The notations A, B, C, D, W, L represent vehicle arrival time, start of service, end of service, departure time, waiting time, and user ride time at a vertices, respectively. For a DARP solution x, let $c(x)$ denote the total travel time of the fleet, and let $q(x), d(x), w(x)$ and $t(x)$ denote the total violation of load, duration, time window and ride time constraints, respectively.

$$q(x) = \sum_{\forall k} \max(q_{k,max} - Q_k, 0), \tag{14}$$

$$d(x) = \sum_{\forall k} \max(T_k' - T_k, 0), \tag{15}$$

$$w(x) = \sum_{\forall i} \big[\max(B_i - l_i, 0) + \max(B_{i+n} - l_{i+n}, 0)\big], \tag{16}$$

$$t(x) = \sum_{\forall i} \max(L_i - L, 0). \tag{17}$$

To set the beginning of service in the best possible way, such that route duration is minimum and ride time limits are respected wherever possible, the route evaluation scheme introduced by Cordeau and Laporte (2003) [19] is applied. It is based on the forward slack time concept of [47] in Eq.18.

$$F_0 = \min_{i\leq j\leq q}\{W_p + (\min_{i<p\leq j}\{l_j - B_j, L - L_j\})\}. \tag{18}$$

The slack at vertex j is the cumulative waiting time until j, plus the minimum of the difference between the end of the time window and the beginning of service at j, and the difference between the maximum user ride time and L_j. The forward time slack at vertex i is the minimum of all slacks between i and q. F_0 thus gives the maximum amount of time by which the departure from the origin depot can be delayed (with equal time window violations and smaller or equal ride time violations) to yield a modified route of minimal duration. Furthermore, Cordeau and Laporte [19] observed that delaying the departure by $\sum_{0<p<q} W_p$ does not affect the arrival time at q, whereas delaying the departure by more will only increase the arrival time at q by as much. Consequently, the departure from the depot should only be delayed by at most $\{F_0, \min_{0<p<q} W_p\}$. The forward time

```
Algorithm 2 GPU-Based Scheduler: dev_scheduler()
Require: Original route $route$, Problem $dev\_problem$
Ensure: Update $routecost$
 1: Create Route $R$ // register memory
    $R \leftarrow route$ // global memory to register memory
    Allot $*A, *B, *C, *D, *W, *L, *q(x_k), *d(x_k), w(x_k), t(x_k)$
       // register memory
    set $D_0 \leftarrow e_0$ // constant memory to register memory
 2: Compute $A_i, w_i, B_i, D_i, y_i$ for each node $i$ in route $k$
    If some $B_i > l_i$ or $y_i > Q$, goto step 8
 3: Compute $F_0$
 4: Set $D_0 = e_0 + min(F_0, \sum_{0<p<q} w_p)$
 5: Update $A_i, w_i, B_i$ and $D_i$
 6: Compute all $T_i^{ride}$
    If all $T_i^{ride} < T^{ride}$ goto step 8 // faster read operation
       since the instance is stored in cached constant memory
 7: For every node $j$ that is an origin
       (a) Compute $F_j$
       (b) Set $w_j := w_j + min(F_j, \sum_{j<p<q} w_p)$;
           $B_j := A_j + w_j$;
           $D_j := B_j + d_j$
       (c) Update $T_i^{ride}$ for each request $i$ whose destination
          is after $j$
       (d) If all $T_i^{ride} <= T^{ride}$ for all $i$ whose destinations
          lie after $j$, return true
    // this step consumes more than two-third of evaluation time
       and it is slower in GPU
      due to the global memory. Therefore, we use the GPU
       register memory whenever
      desired, which is several times faster than the GPU global
       memory.
 8: Compute the value of $routecost$.
```

slack can also be applied to delay B_i at each origin vertex such that the ride time of the corresponding request is reduced. This reasoning led Cordeau and Laporte (2003) [19] to an eight-step evaluation scheme. In their scheme, the steps (1–2) minimize time-window violations $d(x_k)$. In steps (3–6), route duration violations $w(x_k)$ are minimized without increasing time window violations $d(x_k)$. The load violations $q(x_k)$ are irreparable. Step (7) reduces ride-time violations, which is based on the forward slack time technique of [47].

Algorithm 2 is the proposed DARP solution evaluation function, which is based on [19]. The vehicle route is fixed throughout the execution of this algorithm for each input. A vehicle route is defined as a sequence of pickup and dropoff operations. Algorithm 2 takes a vehicle route as its input and produces optimized schedules for the given vehicle route as its output. An optimized schedule is the one that results in the minimum constraint violation for the given vehicle route. We denote M_1 and M_2 as the solution evaluation method of [19] and our Algorithm 2, respectively. M_1 and M_2 have the same scheduling procedure, i.e., both attain the same output for any given input. M_1 and M_2 run on different computational hardware, i.e., M_1 runs in CPU, and M_2 runs in GPU. We have designed M_2 in such a way that it leverages the memory performance of GPU. There are six different GPU memories in the hardware architecture of the Tesla P100 GPU. Each memory architecture has a specific advantage, and our contribution is to map the DARP-related variables to these various memory units. This process of memory-mapping is detailed in Table II, illustrated by Figure 1, and elaborated in Section V-A. In the description of M_2, we have detailed about how the allocation of different DARP-related variables to different GPU memories has improved the computational performance of M_2 in GPU. The role of the GPU in M_2 is to efficiently execute the scheduling algorithm in parallel. In the proposed GPU framework, many such instances of M_2 are run simultaneously in GPU to evaluate the schedules of various vehicle routes during each iteration.

B. Local Search Operation

Next, we present the GPU-accelerated local search algorithm, which will be integrated into the metaheuristics in Section VI. This algorithm is built as a host function that invokes CUDA kernels that run on the GPU. The implementation details are presented in Algorithm 3. This algorithm identifies all neighbors of the current solution, evaluates them, and chooses the best one for the next iteration. In this context, a neighbor refers to a solution generated by performing a neighborhood move onto the current solution, i.e., moving a pair of vertices that belong to a request from its current route to another route. Before launching CUDA kernels, the input arguments (i.e., current solution and DARP instance) are transferred from CPU to GPU; upon launching the kernels, parallel computation is performed in the GPU; finally, the output is copied back to the CPU. The grid size (B), block size (T), stream ID (S), and the GPU shared memory size (M) are initialized according to the current solution, which are detailed in Section V-B. For every request, we launch a set of CUDA kernels to construct and evaluate the neighborhood solutions, namely 1) *Preprocessor*, 2) *Insertion*, 3) *Evaluation*, and 4) *Pipelining*. Algorithms 4-7 describe the implementation of the CUDA kernels.

```
Algorithm 3 GPU-Accelerated Local Search
Require: DARP Instance, Current Solution $x$
Ensure: Crux of Evaluated Neighborhood Routes
 1: Allocate GPU resources.
 2: Initialize $B, T, S, M$.
 3: for all requests $i \in req\_list$ do
 4:     Copy data from CPU to GPU
 5:     Preprocessor Kernel $\ll< B, T, S, M \gg> (gpu[i])$
 6:     Insertion Kernel $\ll< B, T, S, M \gg> (gpu[i])$
 7:     Evaluation Kernel $\ll< B, T, S, M \gg> (gpu[i])$
 8:     Pipelining Kernel $\ll< B, T, S, M \gg> (gpu[i])$
 9:     Copy data from GPU to CPU
10: end for
11: Release GPU resources.
```

C. CUDA Kernels

This section describes the four new CUDA kernels. In the *Preprocessor* kernel function, each thread is responsible to initialize data elements associated with a unique object instantiated with the GPU class. These elements will be used by subsequent kernels to construct and evaluate the solution. The procedure is described as follows: i) initialize the value of the variable idx as a function of threads and blocks, ii) for all values of idx less than N_{size}, where N_{size} is the number of solutions that is in the subset of neighborhood associated with request i, perform step iii, iii) for all nodes in the route $k \in K$, while ensuring the pair-precedence constraints, initialize the values of $start$, gap, $target$, and $routesize$. Algorithm 4 describes the implementation of the *Preprocessor* kernel.

Algorithm 4 Preprocessor Kernel

```
Require: Request $req$, Current Solution $x$
Ensure: Preprocessed $gpu$ objects
 1: int $idx = threadIdx.x + blockIdx.x * blockDim.x$;
 2: if $idx < gpu[idx].N_{size}$ then
 3:     for all ($index$) in all routes $k \in K$ do
 4:         while ensuring the pair-precedence constraints do
 5:             Initialize $gpu[idx].start \leftarrow index$
 6:             Initialize $gpu[idx].gap \leftarrow offset$
 7:             Initialize $gpu[idx].target \leftarrow k$
 8:             Initialize $gpu[idx].routesize \leftarrow x[k].size + 2$
 9:         end while
10:     end for
11: end if
```

In the *Insertion* kernel function, each thread is responsible to construct a unique neighborhood solution. The procedure is described as follows: i) initialize the value of idx as a function of threads and blocks, ii) for all values of idx less than N_{size}, initialize the value of $vehicle$ that represents the target vehicle ID in which the request req will be inserted, iii) for all the values of the variable $tid \in [0, gpu[idx].routesize]$, insert the pick up and drop-off vertices associated with request req in $vehicle$ at the $start$ and gap positions, respectively. As a result, all the neighborhood solutions with respect to the current solution will be constructed. The indexing scheme with $state1$ and $state2$ optimizes branch divergence as detailed in Section V-D. Algorithm 5 describes the implementation of the *Insertion* kernel.

Algorithm 5 Insertion Kernel

```
Require: Request $req$, Current Solution $x$
Ensure: Constructed Neighborhood Routes
 1: int $idx = threadIdx.x + blockIdx.x * blockDim.x$;
 2: bool $state1, state2$; int $node, vehicle$;
 3: if $idx < gpu[idx].N_{size}$ then
 4:     $vehicle \leftarrow gpu[idx].target$
 5:     for $\forall$ $tid \in [0, gpu[idx].routesize]$ do
 6:         $node \leftarrow x[vehicle].route[tid]$
 7:         $state1 \leftarrow tid >= (gpu[idx].start)$
 8:         $state2 \leftarrow tid >= (gpu[idx].start + gpu[idx].gap)$
 9:         $gpu[idx].route[tid + state1 + state2] \leftarrow node$
10:     end for
11: end if
```

In the *Evaluation* kernel function, each thread is responsible to evaluate the constructed neighborhood solutions. The procedure can be described as follows: i) initialize the value of idx as a function of threads and blocks, ii) for all values of idx less than N_{size}, launch the device function $dev\_scheduler$ (Algorithm 2), which schedules the vehicle routes and evaluates the value of the objective function. As described earlier, the Algorithm 2 reduces the constraint violations. The intuition behind separating the *Insertion* and *Evaluation* kernels (Section V-E) is to overlap the launch latency associated with the GPU kernels that belong to different requests. Algorithm 6 describes the implementation of the *Evaluation* kernel.

Algorithm 6 Evaluation Kernel

```
Require: Pointer to $dev\_problem$
Ensure: Evaluated Neighborhood Routes
 1: int $idx = threadIdx.x + blockIdx.x * blockDim.x$;
 2: if $idx < gpu[idx].N_{size}$ then
 3:     $gpu[idx]$.dev_scheduler($dev\_problem$)
 4: end if
```

Algorithm 7 Pipelining Kernel

```
Require: Pointer to $temp\_cost$, Current Solution $x$
Ensure: Crux of Evaluated Neighborhood Routes
 1: int $idx = threadIdx.x + blockIdx.x * blockDim.x$;
 2: if $idx < gpu[idx].N_{size}$ then
 3:     $temp\_cost[idx]$.reset();
 4:     for $\forall k \in [0, m]$ do
 5:         if $k == gpu[idx].target$ then
 6:             $temp\_cost[idx] \leftarrow x[k].routecost$
 7:         else
 8:             $temp\_cost[idx] \leftarrow gpu[idx].routecost$
 9:         end if
10:     end for
11:     $temp\_cost[idx]$.check_feasibility()
12:     $temp[idx].vehicle \leftarrow gpu[idx].target$
13:     $temp[idx].request \leftarrow gpu[idx].request$
14:     $temp[idx].position \leftarrow gpu[idx].position$
15: end if
```

In the *Pipelining* kernel function, each thread is responsible to copy the data from one object to another, which will be transferred to the CPU later. The objective of developing this kernel is to transfer only selective information back to the CPU, thereby reducing the CPU-GPU communication. This procedure is described as follows: i) initialize the value of idx as a function of threads and blocks, ii) for all values of idx less than N_{size}, copy a selective set of information about the solution into an object (i.e., data structure) in the GPU global memory. This selective information contains the value of request ID, current vehicle ID, new vehicle ID, and the positions in which the request will be inserted. Following that, all these objects will be copied from GPU to CPU. The justification for this kernel lies in the fact that the peak bandwidth between the device memory and GPU is much higher than that of the CPU-GPU memory transaction.

Algorithm 7 describes the implementation of the *Pipelining* kernel.

The metaheuristics based on local search can be accelerated by parallel neighborhood exploration using Algorithm 3. The direction of the search can be refined by the metaheuristic procedures in an attempt to identify optimal solutions (Section VI). However, the baseline algorithmic framework does not fully exploit the potential of GPU, impeded by the issues of memory throughput, device occupancy, data transfer latency, thread divergence, and kernel launch latency.

Therefore, we present a set of optimization strategies to circumvent the performance penalties of the GPU framework.

V. Proposed Device Optimization Strategies

A. Device Hierarchical Memory Optimization

The GPU hardware contains various types of memory resources with different sizes and access latencies. The scheduling heuristic of DARP is complex and consists of control-intensive tasks. Therefore, it is important to efficiently map the data structure onto the GPU to exploit its memory bandwidth. The data elements operated by the scheduling heuristic (Algorithm 2) of the GPU class, are classified based on their access type, access frequency, and size. There are data elements that are needed throughout the program, which are stored in the *global memory*. Although the memory latency is the highest, the *global memory* offers a larger capacity for data storage. Besides, the data elements that are no longer required after the execution of a CUDA kernel are mapped onto the *register memory*, which is several hundred times faster than the *global memory*. The spilled data from the *register memory* goes into the *local memory*. *Constant memory* of the GPU is designed to cache data with very low read access latency, henceforth the DARP instance, which is static in nature, is stored in this type of memory. The other types of GPU memory such as *texture* and *shared memory* have their unique advantages, but they are not used in our program. Table II summarizes all the GPU memory types and details how we perform the device hierarchical memory optimization.

Figure 1 illustrates the data mapping onto the GPU, and shows how each GPU thread is mapped to a CUDA core to simultaneously explore a unique solution in the neighborhood of the current solution.

B. Adaptive Kernel Grid Configuration

Kernel grid configuration consists of optimally segregating the tasks with the number of threads and blocks, for efficient parallel computation. To maximize GPU exploitation, it is important to choose the right combination. Since a brute force search may take an enormous amount of time, we develop a kernel-grid heuristic based on the concept of warps. N_{size} defines neighborhood size that can be calculated in advance.

Algorithm 8 shows the kernel-grid heuristic, which is explained as follows: 1) Randomly set grid size B_r and block size T_r, and initialize the values of B_{opt}, T_{opt}, t_{best} and $step_{size}$. Here, the value of $step_{size}$ is set as the warp size, which is 32, thereby avoiding the scenario of branch divergence. 2) For all the values of T in (0,1024] interleaved

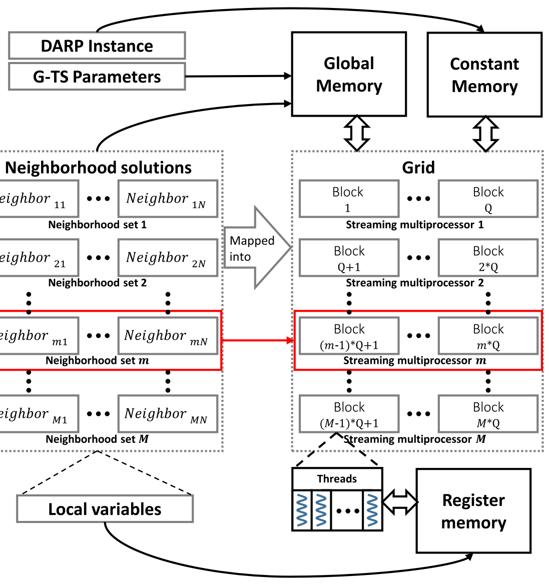


Fig. 1. Memory mapping on the GPU.

Algorithm 8 Adaptive Kernel Grid Configuration

```
$Kernel\_Func \lll B, T, S, W \ggg ()$
$B_r \leftarrow rand(1, 65535)$; $T_r \leftarrow rand(1, 1024)$
$B_{opt} \leftarrow (N_{size}/T_{opt}) + 1$; $T_{opt} = 32 \leftarrow rand(1, 1024)$
Initialize $t_{best} \leftarrow 0$
$step_{size} \leftarrow 32$
for $\forall T \in (0, 1024, stepsize]$ do
    $B \leftarrow round(N_{size}/T) + 1)$
    Launch $Kernel\_Func \lll B, T, S, W \ggg ()$
    Record kernel time $t_{kernel}$
    if $t_{best} \leftarrow 0 \| t_{kernel} < t_{best}$ then
        $t_{best} \leftarrow t_{kernel}$
        $T_{opt} \leftarrow T$
    end if
end for
Set $T \leftarrow T_{opt}$; $B \leftarrow B_{opt}$
```

by the incremental factor $step_{size}$, perform step-3. 3) For each value of T, set the number of blocks B, and launch the kernel functions to record the kernel run time t_{kernel}. Then, update the value of the best number of threads T_{opt} and the best kernel time lapsed t_{kernel}. 4) After a fixed number of iterations, set the grid size as B_{opt} and the block size as T_{opt}. This procedure is capable of attaining a good kernel configuration regardless of the GPU architecture.

C. Optimizing Data Transfer

Whenever a program encounters a CPU-GPU copy instruction in CUDA, the control returns to the CPU only after completing the memory transaction, which creates an unnecessary overhead. Therefore, we have designed the following strategies to optimize data transfer between CPU and GPU: 1) *asynchronous data transfer*: dual direct memory access engine of the GPU allows data transfer over the PCIe bus

TABLE II
DEVICE HIERARCHICAL MEMORY OPTIMIZATION

GPU memory resource			DARP memory mapping			
Memory type	Read access latency	Size	Data mapping	Access type	Access frequency	Size
Global	High	High	Data with program life-time	Read/Write	High	High
Register	Very low	Very low	Data with kernel life-time	Read/Write	Very high	Medium
Local	Mid-high	Medium	Register-spilled data	Read/Write	Very high	Low
Shared	Low	Small	-	-	-	-
Constant	Low (cached)	Medium	DARP Instance	Read only	Medium	Low
Texture	Low (cached)	Medium	-	-	-	-

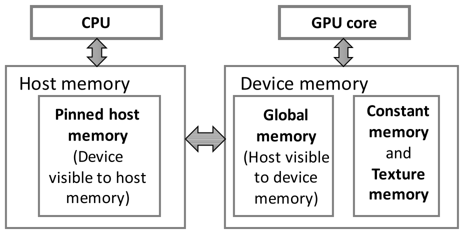

CPU
GPU core
Host memory
Pinned host memory (Device visible to host memory)
Device memory
Global memory (Host visible to device memory)
Constant memory and Texture memory

Fig. 2. Optimized data transfer between CPU and GPU.

from GPU to CPU, while the CUDA cores are busy in crunching the other data. As a result, the program control immediately returns to the next instruction on the CPU side, allowing overlapped execution of both the CPU module and the CPU-GPU data transfer. 2) *batch data transfer*: due to the overhead associated with a single data transfer request, it is preferred to group individual data transfers together to perform contiguous copying. 3) *page-locked data transfer*: host data allocations are pageable by default. After initiating the data transfer, the data elements from pageable memory are moved to pinned memory, before getting transferred to the *global memory*. Therefore, we directly allocate the operating data in the CPU pinned memory.

D. Reducing Control Divergence

In CUDA, each warp comprises of 32 threads that execute a single instruction in one cycle. If multiple instructions are associated with different threads from the same warp, then the executions are serialized. Divergent branches cause inefficient use of computational resources, hence it should be avoided by executing the same set of instructions in all the threads associated with a single warp.

Algorithm 9 illustrates the naive version of the *Insertion* kernel. This CUDA kernel is responsible for the operation of inserting pick up and drop-off vertices that belong to a request i into a route k at positions $start$ and gap, respectively, while initializing other nodes along the route k. The naive idea is to directly implement them using if-else conditions. However, performance penalties may be incurred due to branch divergence. Therefore, we introduce a method to reduce the divergence through boolean operators and an array indexing scheme as described in Algorithm 10. In this method, the boolean variables, $state_1$ and $state_2$, keep track of $start$ and gap, respectively. Finally, the index of the route k for each GPU object is calculated based on the summation of the *state* variables and the current index value i. As a result, the program control does not diverge during the execution of the *Insertion* kernel. The branch refactoring method was first applied for branch and bound algorithm in GPU for flowshop scheduling problem [41]. We extend upon this idea to reduce the control divergence for the application of DARP.

Algorithm 9 Insertion_Kernel (With Control Divergence)
```
for $i = 0;\ i < limit;\ i++$ do
    if $i \leftarrow start$ then
        $gpu[idx].route[i] \leftarrow gpu[idx].start$
    else if $i \leftarrow gap$ then
        $gpu[idx].route[i] \leftarrow gpu[idx].end$
    else
        $gpu[idx].route[i] \leftarrow route[i]$
    end if
end for
```

Algorithm 10 Insertion_Kernel (Without Control Divergence)
```
$gpu[idx].route[i] \leftarrow gpu[idx].start$
$gpu[idx].route[i] \leftarrow gpu[idx].end$
for $i = 0;\ i < limit;\ i++$ do
    bool $state\_1 \leftarrow i >= start$
    bool $state\_2 \leftarrow i >= gap$
    $gpu[idx].route[i + state\_1 + state\_2] \leftarrow route[i]$
end for
```

E. Enhancing Kernel Launch Throughput

For efficient CPU-GPU heterogeneous computing, it is crucial to keep the computational cores busy at all times. In the baseline Algorithm 3, all the kernels operate on a default *stream*. A *stream* is a sequence of operations that execute in issue-order on the GPU. The CPU places issue-orders in the *stream* while retaining the program control. If more than one kernel is launched in a stream, they are serialized. Figure 3 illustrates the proposed method of concurrent kernel execution with kernel slicing. Assume, the kernels in Section IV-C are numbered from 1 to 4. Let $Ka(b)$ indicates the kernel number a placed on stream b. a ranges from 1 to 4 corresponding to the described kernels in Section IV-C; b ranges from 1 to n, where n is the total number of requests. The series of kernels $K1(1), \cdots, K1(n)$ corresponding to stream $1, \cdots, n$ are launched simultaneously. Multiple kernels that use the same stream operate sequentially. Thus, in our implementation, a set of kernels that belongs to the same request is continuously executed regardless of the state of other kernels.

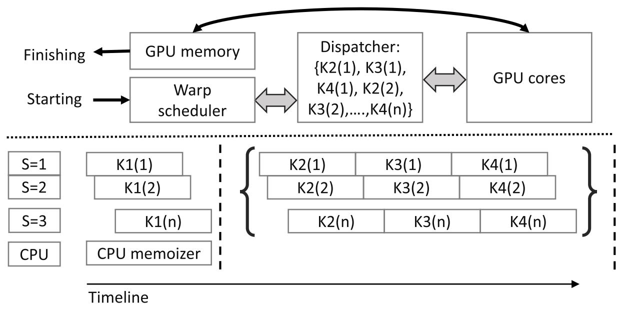


Fig. 3. Concurrent kernel execution with slicing.

VI. Integrating the Proposed GPU Framework With Metaheuristics

In this section, we detail the metaheuristics that are integrated with the GPU framework to solve DARP. Specifically, we explain the neighborhood moves, acceptance criteria to decide the best solution among the neighborhood solutions, allowance of infeasible solutions during the search operation, and implementation details of the metaheuristics such as tabu search (TS) and variable neighborhood search (VNS).

A. Overview of the GPU-Based Metaheuristics

To solve DARP, we iteratively explore the neighborhood solutions of an initial solution until a stopping criterion is met. *DARP instance* will be the input. *Best solution found* is the output, which must be both feasible and should have the least *Cost* or objective function value among all the explored solutions. Firstly, an *Initial solution* for the given DARP instance will be constructed and it will be set as the *Current solution*. *Current solution* is the solution present at the current iteration. Then, the process of iteratively exploring the neighborhood of a *Current solution* is performed using GPU. The ways to generate *neighborhood solutions* from the *Current solution*, and acceptance criterion to select the best admissible solution on each iteration are based on the integrated metaheuristic. During each iteration, the best admissible solution among the *neighborhood solutions* is selected in CPU. For search diversification, the acceptance criteria may allow infeasible solutions to be selected. However, the *Best solution found* must be always feasible and updated after every iteration.

B. Neighborhood Move

The process of moving a customer request from one vehicle into another is called as *neighborhood move*. All possible solutions resulting from moving a request into all available vehicles are called as *neighborhood solutions*. For GPU computation, the neighborhood solutions are divided into batches that are executed concurrently in GPU. Each batch simultaneously evaluates the process of moving a unique request into all vehicles. Inside each batch, the GPU threads simultaneously evaluate the process of moving a unique request into a unique vehicle. The choice of which customer requests need to be relocated is chosen by the integrated metaheuristic.

C. Acceptance Criteria to Iteratively Select Best Solution

There are two strategies for iteratively accepting the best solution among the neighborhood namely, first-improvement and best-improvement. In the first-improvement strategy, the neighborhood solutions are evaluated one-by-one, once a better solution is found, the control will move to the next iteration. In the best-improvement strategy, all the neighborhood solutions need to be evaluated and the best one will be found before proceeding to the next iteration. We have decided to use the best-improvement strategy based on the following reasons: 1) complete neighborhood exploration has the maximum potential to find the best solution during the search [48], and 2) since we use GPU that has thousands of cores, it is faster to evaluate all the neighborhood solutions simultaneously.

D. Allowance of Infeasible Solutions

In order to escape local optima, infeasible solutions are allowed during the search process through a relaxation mechanism [19]. Solutions are evaluated by a modified objective function $f_{new}(x) = f(x) + \alpha' * q(x) + \beta' * d(x) + \gamma' * w(x) + \phi' * t(x)$, where $\alpha', \beta', \gamma', \phi'$ are self-adjusting positive parameters. During every iteration, these parameters are dynamically increased and decreased by a factor of $(1 + \delta')$, if the current solution is feasible and infeasible, respectively. A DARP solution is feasible if it does not violate any given constraints in the formulation. We use the same parameters and relaxation mechanism described in [19] to facilitate the exploration of solution space.

E. Implementation Details

1) Tabu Search Heuristic: Tabu search heuristic was first introduced in [49], and applied to solve DARP in [19]. It works based on the concept of forbidding recently visited neighborhood moves for a limited number of iterations to avoid getting trapped at local optimal solutions. It is often regarded as a higher-level procedure that guides lower level heuristics. In our implementation, the lower level heuristic is the GPU-accelerated local search method. Therefore, we extend the tabu search of [19] with the proposed GPU framework to form G-TS. We have reported the preliminary results in [46].

In tabu search, all requests are removed and reinserted iteratively. A request that is removed from a route must be reinserted in a different route. During the search, we allow infeasible solutions as described by [19]. In TS, recently performed neighborhood moves will be updated in an array consisting of *tabu moves*. Such moves cannot be repeated for a subsequent number of iterations. During the process of selecting the best admissible solution in a neighborhood, *tabu moves* are not considered. If none of the neighborhood solutions is feasible, then the best infeasible solution is selected as the solution that has the least value of the objective function. However, if any neighborhood solution is better than the *Best solution found*, then the *tabu move* can be overridden.

Algorithm 11 describes the implementation of our GPU-based tabu search algorithm to solve the DARP. The procedure can be described as follows: 1) allocate device resources, and initialize the *tabu coefficients* (relaxation mechanism) based on [19], 2) construct initial solution (as per Section VII), 3) copy *Current solution* from CPU

Algorithm 11 Algorithm: GPU-Based TS

```
Require: DARP Instance
Ensure: Best solution found and $Cost_{best}$
 1: Parameterization and allocation of GPU resources.
 2: Copy DARP instance from CPU to GPU.
 3: Construct an Initial solution in CPU
 4: Set Current solution as Initial solution in CPU
 5: repeat
 6:     Update $req\_list$ based on TS in CPU
 7:     Copy Current solution from CPU to GPU
 8:     // Evaluating neighborhoods in multiple concurrent
        batches
 9:     Explore neighborhood in GPU i.e. launch
        GPU-accelerated Local Search()
10:     Selectively copy data from GPU to CPU
11:     Update the overall cost of each neighborhood solution
        in CPU
12:     Select best solution among the neighborhood solutions
        based on TS in CPU
13:     Set the newly selected solution as Current solution in
        CPU
14:     Update $Cost_{best}$ of the Best solution found in CPU
15:     Adjust tabu coefficients (relaxation mechanism)
16: until stopping criterion
17: Release GPU resources.
18: return Best solution found
```

Algorithm 12 Algorithm: GPU-Based VNS

```
Require: DARP Instance
Ensure: Best solution found and $Cost_{best}$
 1: Parameterization and allocation of GPU resources.
 2: Copy DARP instance from CPU to GPU.
 3: Construct an Initial solution in CPU
 4: Set Current solution as Initial solution in CPU
 5: repeat
 6:     Update $req\_list$ based on VNS in CPU
 7:     Copy Current solution from CPU to GPU
 8:     // Evaluating neighborhoods in a single batch
 9:     Explore neighborhood in GPU i.e. launch
        GPU-accelerated Local Search()
10:     Selectively copy data from GPU to CPU
11:     Update the overall cost of each neighborhood solution
        in CPU
12:     Select best solution among the neighborhood solutions
        based on VNS in CPU
13:     Set the newly selected solution as Current solution in
        CPU
14:     Update $Cost_{best}$ of the Best solution found in CPU
15:     if No improvement in $Cost_{current}$ for a predefined
        number of iterations then
16:         Set the Current solution as the Best solution found
17:     end if
18: until stopping criterion
19: Release GPU resources.
20: return Best solution found
```

to GPU, 4) construct and evaluate neighborhood solutions in GPU, 5) selectively copy data from GPU to CPU, 6) identify the best admissible solution and set it as the *Current solution* for next iteration, 7) repeat steps 3-6 until the stopping criterion is fulfilled, and 9) output the *Best solution found* and release device resources.

2) Variable Neighborhood Search: In general, VNS starts from an initial solution. Then, during every iteration, a solution x' is selected from a set of neighborhood solutions of a *Current solution* x by shaking procedure. A subsequent local search step may be applied to the selected solution x' to yield x''. If x'' is better than x, it replaces x and the search continues with the first neighborhood $k = 1$. If x'' is worse, x is not replaced and the next (larger) neighborhood is used in the subsequent iteration $k = k + 1$. Whenever k reaches the maximum number of neighborhoods k_{max}, the search continues with the first neighborhood $k = 1$. This is repeated until the stopping criterion is met.

Previously, VNS was applied to DARP in [20] which has complicated shaking procedures and design decisions. Instead, we employ the two simpler shaking procedures from the literature namely, 1) ejection neighborhood (random and worst) [21], and 2) zero-split neighborhood [20]. During every iteration, we randomly choose the type of shaking procedure. In the ejection neighborhood, a set of requests is chosen, removed from the current solution, and reinserted at the best possible position among all vehicles. The requests can be removed in two ways: i) random removal of requests, and ii) those requests that most degrade the value of the objective function can be removed. The number of neighborhoods, i.e., requests to be relocated k varies from 1 to 7. In zero-split neighborhood, a sequence of requests is moved from one vehicle to another. A request that is removed from its current route can be reinserted into any route. Sometimes, sequences of requests lie between two arcs where the vehicle load is equal to zero, i.e., the vehicle is empty. Every vehicle may contain such multiple sequences of requests. In this neighborhood, a randomly chosen such sequence of requests will be removed from its current vehicle. All requests that are part of this sequence are then reinserted one-by-one at the best position into the vehicles. A thresholding criterion is employed to accept the solution resulting from the shaking process. The local search step is carried out only when a new value of the *Best solution found* is found in that iteration. During the local search phase, each request is removed from its corresponding route and reinserted at the best possible positions of the same route. Additionally, we employ a restart mechanism, in which, if the *Current solution* does not improve for more than 200 iterations, we set the *Current solution* as the *Best solution found* and restart the search for intensification. We allow the infeasible solutions to be accepted during the optimization process for diversification.

Algorithm 12 describes the implementation of our GPU-based VNS method for DARP. The procedure can be described as follows: 1) allocate device resources, 2) construct initial solution (as per Section VII), 3) copy *Current solution*

TABLE III
REPRESENTATION OF THE METHODOLOGIES

Notation	Description
TS	Described tabu search heuristic methodology
VNS	Described variable neighborhood search methodology
C_1	TS with conventional GPU methodology
C_1+opt_1	TS with conventional GPU methodology with device hierarchical memory optimization in section V-A
C_1+opt_2	TS with conventional GPU methodology with adaptive kernel grid configuration in section V-B
C_1+opt_3	TS with conventional GPU methodology with optimizing data transfer in section V-C
C_1+opt_4	TS with conventional GPU methodology with optimizing thread divergence in section V-D
C_1+opt_5	TS with conventional GPU methodology with concurrent kernels in section V-E
P_1	TS with proposed GPU methodology (described GPU framework + all optimization strategies in section V)
P_1-opt_1	TS with proposed GPU methodology without device hierarchical memory optimization in section V-A
P_1-opt_2	TS with proposed GPU methodology without adaptive kernel grid configuration in section V-B
P_1-opt_3	TS with proposed GPU methodology without optimizing data transfer in section V-C
P_1-opt_4	TS with proposed GPU methodology without optimizing thread divergence in section V-D
P_1-opt_5	TS with proposed GPU methodology without concurrent kernels in section V-E
C_2	VNS with conventional GPU methodology
C_2+opt_1	TS with conventional GPU methodology with device hierarchical memory optimization in section V-A
C_2+opt_2	VNS with conventional GPU methodology with adaptive kernel grid configuration in section V-B
C_2+opt_3	VNS with conventional GPU methodology with optimizing data transfer in section V-C
C_2+opt_4	VNS with conventional GPU methodology with optimizing thread divergence in section V-D
C_2+opt_5	VNS with conventional GPU methodology with concurrent kernels in section V-E
P_2	VNS with proposed GPU methodology (described GPU framework + all optimization strategies in section V)
P_2-opt_1	VNS with proposed GPU methodology without device hierarchical memory optimization in section V-A
P_2-opt_2	VNS with proposed GPU methodology without adaptive kernel grid configuration in section V-B
P_2-opt_3	VNS with proposed GPU methodology without optimizing data transfer in section V-C
P_2-opt_4	VNS with proposed GPU methodology without optimizing thread divergence in section V-D
P_2-opt_5	VNS with proposed GPU methodology without concurrent kernels in section V-E
E_1	Existing results based on VNS reported by Parragh et al. [20]
E_2	Existing results based on TS of Cordeau and Laporte (2003) [19] reported by Kirchler et al. [31]
E_3	Existing results based on Granular TS variant, GTS_P reported by Kirchler et al. [31]
E_4	Existing results based on Granular TS variant, GTS_RC reported by Kirchler et al. [31]

from CPU to GPU, 4) construct and evaluate neighborhood solutions in GPU, 5) selectively copy data from GPU to CPU, and select best admissible solution in CPU, 6) trigger restart mechanism in CPU if required, 7) record the cost of best solution $Cost_{best}$ in CPU, 8) repeat steps 3-7 until the stopping criterion, 9) print the *Best solution found*, and release the memory resources.

In the next section, we evaluate the performance of the proposed GPU-based solution methodology and compare the results with the existing approaches for solving DARP.

VII. PERFORMANCE EVALUATION

A. Simulation Settings

The GPU-accelerated algorithms are implemented in CUDA C and conventional CPU-based algorithms are implemented in C++ language. All simulations have been carried out on a computer running 2.1 GHz Intel Xeon E5-2620 v4 processor and NVidia Tesla P100-PCI-E-16GB GPU, interconnected with a PCI-e controller. We have used Visual studio 2015 with CUDA 8.0 toolkit in Windows 8.1 as the integrated development environment.

The DARP test instances are obtained from [19]. In these instances, the number of requests ranges from 24 to 144, and the number of vehicles ranges from 3 to 13. Each request is randomly generated in the Euclidean space of $[-10, 10]^2$. For R1a-R10a, the time window length is 15 min. For R1b-R10b, the time window length is 45 min. Each instance has a different geographical distribution of pick-up and drop-off locations. The service time for each vehicle at any location is set as 10 min. The capacity of each vehicle is 6, with a maximum route duration limit of 480 min. The maximum passenger ride time limit is 90 min. All vehicles are considered to travel at a constant speed in the road network.

To ensure a fair comparison between CPU and GPU methods, all algorithms start with an initial solution generated by the same construction heuristic, described as follows: i) create K set of of empty routes, ii) for each route $k \in K$, randomly initialize the maximum number of pick up and drop-off vertices that can be inserted, iii) create a req_list that comprises n requests in a random order, iv) for each unserved request $i \in req\_list$, randomly insert its pick up and drop-off vertices into any route $k \in K$ while ensuring the pair-precedence constraints.

B. Effect of Different Components of the Proposed GPU Methodology on Computational Speedup

Table III describes the notations of the various solution methodologies. The proposed GPU algorithms are represented by $P \in P_1, P_2$ series, conventional GPU methodologies are denoted by $C \in C1, C2$ series, and existing best-performing sequential methods are mentioned by $E \in E_1, E_2, E_3, E_4$ series. The device optimization strategies are represented by $opt_1, opt_2, opt_3, opt_4, opt_5$, where opt_1 denotes device hierarchical memory optimization, opt_2 denotes adaptive kernel grid configuration, opt_3 denotes data transfer optimization, opt_4 denotes reducing thread divergence, opt_5 denotes concurrent kernel mechanism. For example, $P - opt_1$ represents the

TABLE IV
EFFECT OF DIFFERENT COMPONENTS ON COMPUTATIONAL SPEEDUP FOR THE PROPOSED GPU-BASED TABU SEARCH METHODOLOGY

Case	n	m	Speedup: GPU-based TS methodology											
			C_1	C_1+opt_1	C_1+opt_2	C_1+opt_3	C_1+opt_4	C_1+opt_5	P_1-opt_1	P_1-opt_2	P_1-opt_3	P_1-opt_4	P_1-opt_5	P_1
R1a	24	3	2.89	3.62	4.04	3.52	2.99	11.68	20.63	19.11	4.43	25.71	5.75	**26.62**
R2a	48	5	4.95	6.95	9.44	6.03	5.29	21.67	34.95	35.51	10.85	45.29	13.48	**51.44**
R3a	72	7	7.14	11.37	13.70	8.30	7.62	26.25	33.78	48.73	17.20	61.50	21.16	**67.66**
R5a	120	11	11.87	18.65	18.59	13.07	12.82	28.71	29.87	57.94	26.06	63.11	29.78	**68.22**
R9a	108	8	12.32	19.19	19.77	14.04	13.14	24.15	26.62	58.58	29.26	65.70	33.73	**65.97**
R10a	144	10	16.98	25.39	23.46	18.81	17.51	23.71	25.19	51.82	34.65	53.44	39.84	**55.55**
R1b	24	3	2.29	2.92	3.34	2.83	2.46	9.10	17.76	13.93	3.66	19.60	4.69	**21.07**
R2b	48	5	4.71	6.57	9.00	5.66	5.18	19.85	29.39	33.40	10.20	43.97	12.41	**47.37**
R5b	120	11	12.32	18.24	19.79	14.06	13.26	27.47	30.20	62.60	27.22	65.52	31.13	**66.68**
R6b	144	13	14.91	21.85	22.31	16.18	15.25	27.19	30.32	61.09	31.13	66.24	34.87	**66.63**
R7b	36	4	3.48	5.01	6.20	4.11	3.74	14.83	25.43	22.83	7.02	34.04	8.78	**35.56**
R9b	108	8	15.96	24.07	25.63	17.60	17.17	33.70	36.45	81.00	37.12	90.05	42.09	**92.63**
R10b	144	10	16.53	25.13	23.43	18.93	17.53	24.14	25.28	52.69	34.77	53.48	39.68	**56.21**
Avg.	88	8	9.72	14.54	15.29	11.01	10.30	22.50	28.14	46.10	21.04	52.90	24.41	**55.51**

TABLE V
EFFECT OF DIFFERENT COMPONENTS ON COMPUTATIONAL SPEEDUP FOR THE PROPOSED GPU-BASED VNS METHODOLOGY

Case	n	m	Speedup: GPU-based VNS methodology											
			C_2	C_2+opt_1	C_2+opt_2	C_2+opt_3	C_2+opt_4	C_2+opt_5	P_2-opt_1	P_2-opt_2	P_2-opt_3	P_2-opt_4	P_2-opt_5	P_2
R1a	24	3	3.97	5.06	5.35	4.49	4.34	4.72	7.54	8.55	6.69	7.99	8.09	**10.19**
R2a	48	5	4.69	7.31	6.46	5.68	5.75	5.88	6.41	6.74	6.88	7.56	7.35	**12.06**
R3a	72	7	8.24	9.77	10.57	10.32	8.88	13.06	8.96	10.90	9.03	11.48	11.59	**15.33**
R5a	120	11	11.34	13.40	14.01	14.55	15.22	14.40	7.01	9.04	7.71	7.26	10.28	**17.76**
R9a	108	8	21.58	29.45	30.24	24.51	22.32	24.44	36.56	41.31	41.26	44.74	41.88	**48.12**
R10a	144	10	27.65	33.18	32.89	39.46	28.44	29.85	18.87	18.50	10.51	25.88	24.36	**50.81**
R1b	24	3	4.53	6.41	6.46	5.74	4.93	5.98	8.57	9.38	8.95	10.29	10.79	**11.95**
R2b	48	5	5.60	6.73	7.92	6.78	6.27	7.00	8.00	10.13	8.41	10.90	10.20	**14.99**
R5b	120	11	6.93	11.04	9.77	7.55	9.35	9.60	6.19	7.87	6.30	8.96	7.22	**15.33**
R6b	144	13	8.40	10.11	11.66	10.00	11.19	13.04	5.15	5.71	6.05	7.18	5.34	**16.92**
R7b	36	4	5.67	7.46	9.04	6.87	6.70	6.96	10.02	11.86	9.44	11.99	12.46	**16.42**
R9b	108	8	9.50	12.44	12.09	12.97	12.79	13.30	6.12	7.39	7.05	8.37	9.10	**15.58**
R10b	144	10	28.95	34.53	32.66	31.06	29.13	30.88	40.36	22.12	21.08	35.05	21.59	**52.73**
Avg.	88	8	11.31	14.38	14.55	13.85	12.72	13.78	13.06	13.04	11.49	15.20	13.87	**22.94**

proposed GPU methodology without optimization strategy opt_1. In another example, $C + opt_1$ represents the conventional GPU methodology with the optimization strategy opt_1. The conventional GPU methodology with all the five device optimization strategies is equivalent to the proposed GPU methodology. $C_1 + opt_1, opt_2, opt_3, opt_4, opt_5$ is equivalent to P_1. $C_2 + opt_1, opt_2, opt_3, opt_4, opt_5$ is equivalent to P_2.

In terms of computational speedup, the effect of including each device optimization strategy into the conventional GPU methodology, and the effect of excluding each device optimization strategy from the proposed GPU methodology are shown in Tables IV and V. Initially, the computational time for the GPU-based methods and their CPU counterparts on DARP instances are recorded after 10^2 number of iterations. Then, the computational speedup is calculated as follows.

$$speedup = \frac{Execution\ time\ of\ CPU\ only\ method}{Execution\ time\ of\ GPU\text{-}based\ method}. \quad (19)$$

Each experiment is repeated five times to calculate the average speedup. The fundamental algorithmic behavior of the GPU-based methods and their CPU counterparts remains the same except for the computational speed. The existing methods $E \in E_1, E_2, E_3, E_4$ provide results for 13 test instances. For the remaining 7 instances, the detailed information about the solution at different computation times are not available. Therefore, we also restrict our tests to those 13 test instances.

With the conventional GPU methodologies C, the inclusion of strategies opt_1, opt_2, opt_5 leads to the largest increase in speedup, and the inclusion of strategy opt_4 leads to the lowest increase in speedup. With the proposed GPU methodologies P, the exclusion of strategy opt_3 leads to the largest decrease in speedup, and exclusion of strategy opt_4 leads to the smallest decrease in speedup. Therefore, when designing a GPU-based algorithm for DARP, our results suggest the following: 1) it is preferred to first include the strategies opt_1, opt_2, opt_5 followed by opt_3, opt_4, 2) it is most advisable to remove the strategy opt_4 and least preferable to remove strategy opt_3 from the GPU methodology during the process of algorithmic simplifications. Furthermore, it can be seen that speedup is the largest when all device optimization strategies ($opt_1, opt_2, opt_3, opt_4, opt_5$) are included in the GPU methodology. As a result, the GPU-based tabu search algorithm P_1 and GPU-based variable neighborhood search P_2 exhibit the highest computational speedup. Therefore, we have considered only these two methods for the subsequent experiments.

TABLE VI
COMPARISON OF SOLUTION QUALITY (*gap*) OF OUR APPROACH (P) AND EXISTING METHODS (E) AFTER 30 S, 60 S AND 180 S OF CPU TIME

Case	*BKS*	30 sec						60 sec						180 sec					
		E_1	E_2	E_3	E_4	P_1	P_2	E_1	E_2	E_3	E_4	P_1	P_2	E_1	E_2	E_3	E_4	P_1	P_2
	f'																		
R1a	3152.22	0.13	0.48	2.66	1.02	**0.00**	**0.00**	0.13	0.48	2.66	0.84	**0.00**	**0.00**	0.04	0.48	2.67	0.48	**0.00**	**0.00**
R2a	6027.29	151.51	29.79	5.07	7.52	**-3.16**	**-3.64**	154.01	11.72	5.07	5.30	**-3.12**	**-3.80**	116.09	3.46	3.37	1.71	**-4.29**	**-4.24**
R3a	9952.00	50.25	1.45	17.12	1.74	**-1.74**	**0.17**	56.46	0.62	7.86	1.23	**-2.45**	**-1.06**	50.21	0.13	5.08	0.10	**-2.55**	**-1.76**
R5a	13050.30	123.61	67.57	169.20	8.34	**-1.88**	**0.98**	93.20	24.28	129.97	6.09	**-2.60**	**0.09**	68.90	3.01	50.34	1.76	**-2.82**	**-0.72**
R9a	13912.96	Inf	91.33	168.93	50.81	**11.79**	**21.09**	**18.11**	46.11	82.21	50.81	**10.68**	19.23	12.38	29.01	21.98	48.88	**6.32**	**6.55**
R10a	18259.40	Inf	33.92	101.04	14.94	**6.63**	**19.42**	Inf	19.60	94.47	**9.93**	**-2.84**	12.26	35.67	11.59	75.56	5.84	**-4.42**	**5.40**
R1b	2874.74	0.00	1.44	1.13	2.80	**0.00**	**0.00**	0.00	1.44	1.13	1.18	**0.00**	**0.00**	0.00	1.18	1.13	1.18	**0.00**	**0.00**
R2b	4969.28	3.47	0.35	17.95	1.98	**-0.65**	**-0.26**	1.68	0.35	4.80	1.62	**-1.12**	**-0.35**	-0.48	0.35	3.43	1.29	**-1.15**	**-0.58**
R5b	11747.20	33.31	90.71	109.01	8.59	**0.48**	**1.62**	26.96	59.08	98.36	6.90	**-1.29**	**1.33**	15.60	13.83	50.90	3.95	**-1.73**	**0.35**
R6b	15000.50	51.57	100.24	127.68	8.61	**0.09**	**1.90**	45.86	78.62	114.07	5.77	**-1.33**	**1.52**	10.42	36.55	104.63	2.57	**-1.68**	**0.24**
R7b	4365.98	5.72	0.49	5.08	1.06	**-1.10**	**-0.61**	5.35	0.00	5.08	1.06	**-1.40**	**-1.18**	4.80	0.00	5.08	1.06	**-1.40**	**-1.40**
R9b	12265.50	30.26	46.32	84.76	4.68	**-0.54**	**1.45**	33.78	17.62	74.83	1.92	**-1.80**	**0.88**	7.44	2.61	65.38	1.60	**-2.04**	**-0.64**
R10b	16391.00	31.30	190.87	232.66	34.05	**5.22**	**12.57**	19.73	69.52	224.47	26.57	**-1.20**	**9.44**	19.86	10.00	191.33	6.12	**-2.14**	**4.68**
	f''																		
R1a	3118.33	0.14	0.91	0.38	1.23	**0.00**	**0.00**	0.14	0.00	0.02	1.23	**0.00**	**0.00**	0.05	0.00	0.00	0.21	**0.00**	**0.00**
R2a	5546.55	1.38	3.29	4.78	3.05	**-0.03**	**1.09**	0.99	3.29	3.58	2.11	**-0.20**	**1.09**	-0.62	1.50	2.38	1.99	**-0.34**	**0.00**
R3a	9632.47	2.91	3.66	23.85	3.91	**0.53**	**1.56**	1.82	2.12	8.60	3.10	**-0.01**	**1.01**	0.39	1.86	3.00	3.10	**-0.16**	**0.57**
R5a	12642.77	11.47	59.16	64.40	7.91	**1.10**	**2.83**	9.02	18.83	55.39	6.00	**0.01**	**1.92**	5.42	2.75	28.14	3.59	**-0.32**	**1.16**
R9a	13301.65	Inf	83.04	101.83	50.78	**10.10**	**28.59**	18.26	81.41	51.40	34.99	**7.67**	**26.54**	**8.25**	29.09	47.74	27.97	**7.67**	9.09
R10a	17459.16	Inf	37.74	105.62	12.09	**5.91**	**10.42**	Inf	13.18	95.96	10.27	**-0.40**	**7.28**	12.53	7.54	81.76	5.50	**-1.51**	**4.12**
R1b	2874.74	0.00	1.35	2.67	3.48	**0.00**	**0.00**	0.00	1.35	2.67	3.48	**0.00**	**0.00**	0.00	1.07	2.67	1.46	**0.00**	**0.00**
R2b	4929.08	1.32	2.65	16.14	2.63	**0.09**	**0.58**	1.02	2.65	4.43	2.63	**-0.42**	**-0.12**	0.33	0.94	1.78	1.97	**-0.48**	**-0.35**
R5b	11635.79	11.82	83.45	66.09	7.63	**1.24**	**2.58**	9.43	54.02	59.64	6.01	**-0.85**	**1.80**	5.18	10.60	51.22	2.96	**-0.98**	**0.60**
R6b	14927.1	12.00	91.74	69.22	10.05	**0.81**	**1.95**	10.29	72.22	64.46	5.96	**-0.88**	**1.29**	7.13	31.82	54.32	3.86	**-1.43**	**0.49**
R7b	4290.11	0.83	1.82	2.84	0.47	**-0.53**	**-0.36**	0.45	1.82	2.84	0.38	**-0.60**	**-0.48**	-0.17	0.21	2.14	0.38	**-0.60**	**-0.60**
R9b	12021.8	10.19	48.52	78.77	7.39	**1.46**	**1.64**	8.39	19.60	76.41	3.78	**0.11**	**1.24**	4.92	4.23	61.81	3.39	**-0.12**	**0.54**
R10b	16238.27	20.25	148.07	235.76	45.83	**4.02**	**11.30**	16.07	67.18	207.99	45.65	**0.82**	**7.43**	10.65	8.58	193.24	10.82	**-0.91**	**2.96**
Avg. *gap*	0.00	Inf	46.94	69.79	11.64	**1.53**	**4.50**	Inf	25.66	56.86	9.42	**-0.12**	**3.36**	15.19	8.17	42.73	5.53	**-0.66**	**1.02**

C. Comparison of Solution Quality of Our Approach With the Existing Results

Table VI compares the solution quality of the proposed GPU approaches $P \in P_1, P_2$, in terms of *gap* defined in Eq. 20, with the best-performing sequential approaches $E \in E_1, E_2, E_3, E_4$ from the literature. For each instance, two of the best values of *gap* are highlighted in bold text. Each experiment is conducted five times, and average values of *gap* are reported. For uniformity, the objective function values reported in E are converted into gap values in the Table VI. The definition for *gap* is given by:

$$gap\ (\%) = \frac{Cost - BKS}{BKS} \times 100, \tag{20}$$

where *BKS* denotes the best-known solutions or objective function value, and *Cost* denotes the best value of objective function attained for a given solution method. The solution quality is measured by the value of *gap*. Lesser the value of *gap* represents a better solution quality. A negative value of *gap* denotes a quality of solution better than *BKS*. We attempt to minimize the value of *gap*. To consider DARP under dynamic real-life setting, Kirchler and Calvo (2013) [31] has reported the solution quality at 30 sec, 60 sec, and 180 sec. We applied the same CPU time for the optimization process. We report solution quality for two objective functions $f' = f$, and $f'' = f' - w_5 * e(s)$, and used same weights for the objective function f as proposed in [20], [25], [31]: $w_1 = 8, w_2 = 3, w_3 = 1, w_4 = 1, w_5 = n$, and the coefficient α was set to 10000. *BKS* for all instances with objective functions f' and f'' are provided by [20], [31].

Since we do not find existing GPU-based results for DARP, we compare the solution quality of our approach P_1, P_2 with the best-performing sequential approaches E_1, E_2, E_3, E_4. From Table VI, the proposed GPU methodologies P_1 and P_2 attain better solutions than E for all the tested instances, and better than *BKS* for most of the instances within a short amount of time. Furthermore, P_1 outperforms P_2 in terms of average *gap* over the tested instances. In P_2, the process of inserting selected customers into all vehicles is evaluated using GPU in a single batch. In a single batch, multiple GPU threads will be running simultaneously, in which each GPU thread evaluates the process of inserting a unique customer into a unique vehicle. In P_1, the processes of inserting each customer into all vehicles are evaluated using GPU in multiple concurrent batches, i.e., the process of inserting any customer into all vehicles is parallelized with concurrent kernels, therefore resulting in rapid exploration of a much larger neighborhood. For all tested instances, we have provided the *BKS* and *gap* values attained by the methods for both the objective function f' and f'' for the sake of comparison. In summary, the proposed methodologies outperform the existing methods in terms of solution quality within a short time.

VIII. CONCLUSION

On-demand mobility has become a topic of interest due to its wide range of applications including medical emergency transit, public transportation, airport, and seaport operations. In the last few years, the general public has developed a significant interest over MoD transportation. Dial-a-ride problem is a form of MoD, which is NP-hard in nature. In today's world, DARP often occurs in "real-time" rather than in advanced reservation mode, and hence attaining quicker solutions can be critical. GPU computing has turned out to be a fundamental pillar in modern science. In this paper, we have proposed a GPU-based solution methodology to solve DARP under dynamic real-life settings. First, we have introduced the building blocks of the proposed GPU framework including CUDA kernels and local search operation. Secondly, we have proposed five device optimization strategies tailored for the described problem in order to improve computational speedup using GPU. Thirdly, we have integrated the proposed framework with two metaheuristics such as tabu search and variable neighborhood search. Simulations are conducted on DARP benchmark instances from the literature. The proposed GPU approach with the device optimization strategies attains higher speedup when compared to the conventional GPU approach. The effect of inclusion and exclusion of each device optimization strategy on computational speedup has been analyzed. Overall, the proposed GPU approach attains better average solution quality when compared to the existing best-performing sequential approaches for DARP in a short time. We conclude that accelerating time-critical neighborhood explorations in GPU along with a guided search of metaheuristic can be significant in efficiently solving DARP.

Following are some potential future research directions: 1) introducing GPU-based neighborhood reduction techniques to address very large scale problems, 2) solving the richer variants of DARP with heterogeneous demand, fleet, dynamic traffic conditions, 3) applying the proposed framework for emergency transportation services (e.g. medical ambulance, fire and rescue operations), where attaining quick solutions is critical, 4) enabling the GPU memory coalescing techniques, 5) utilizing multi-core CPU alongside the proposed GPU framework to further enhance the computational speedup.

Acknowledgment

The authors thank the anonymous reviewers and associate editor for their valuable comments and suggestions which helped to significantly improve the quality of this manuscript.

References

[1] M. Pavone, "Autonomous mobility-on-demand systems for future urban mobility," in *Autonomes Fahren*. Berlin, Germany: Springer Vieweg, 2015, pp. 399–416.

[2] C. E. Borroni-Bird, L. D. Burns, and W. J. Mitchell, *Reinventing the Automobile: Personal urban mobility for the 21st century*. Cambridge, MA, USA: MIT Press, 2010.

[3] G. B. Dantzig and J. H. Ramser, "The truck dispatching problem," *Manage. Sci.*, vol. 6, no. 1, pp. 80–91, Oct. 1959.

[4] S. C. Ho, W. Y. Szeto, Y.-H. Kuo, J. M. Y. Leung, M. Petering, and T. W. H. Tou, "A survey of dial-a-ride problems: Literature review and recent developments," *Transp. Res. B, Methodol.*, vol. 111, pp. 395–421, May 2018.

[5] H. N. Psaraftis, M. Wen, and C. A. Kontovas, "Dynamic vehicle routing problems: Three decades and counting," *Networks*, vol. 67, no. 1, pp. 3–31, Jan. 2016.

[6] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "NVIDIA tesla: A unified graphics and computing architecture," *IEEE Micro*, vol. 28, no. 2, pp. 39–55, Mar. 2008.

[7] T. V. Luong, N. Melab, and E.-G. Talbi, "GPU computing for parallel local search Metaheuristic algorithms," *IEEE Trans. Comput.*, vol. 62, no. 1, pp. 173–185, Jan. 2013.

[8] N. H. Wilson, J. Sussman, H.-K. Wong, and T. Higonnet, *Scheduling Algorithms for a Dial-a-Ride System*. Cambridge, MA, USA: Massachusetts Institute of Technology, 1971.

[9] P. Toth and D. Vigo, "Heuristic algorithms for the handicapped persons transportation problem," *Transp. Sci.*, vol. 31, no. 1, pp. 60–71, Feb. 1997.

[10] O. B. G. Madsen, H. F. Ravn, and J. M. Rygaard, "A heuristic algorithm for a dial-a-ride problem with time windows, multiple capacities, and multiple objectives," *Ann. Oper. Res.*, vol. 60, no. 1, pp. 193–208, Dec. 1995.

[11] R. W. Calvo and A. Colorni, "An effective and fast heuristic for the Dial-a-Ride problem," *4OR*, vol. 5, no. 1, pp. 61–73, Mar. 2007.

[12] H. N. Psaraftis, "An exact algorithm for the single vehicle Many-to-Many Dial-A-Ride problem with time windows," *Transp. Sci.*, vol. 17, no. 3, pp. 351–357, Aug. 1983.

[13] J. Desrosiers, Y. Dumas, and F. Soumis, "A dynamic programming solution of the large-scale single-vehicle Dial-A-Ride problem with time windows," *Amer. J. Math. Manage. Sci.*, vol. 6, nos. 3–4, pp. 301–325, Feb. 1986.

[14] T. R. Sexton and Y.-M. Choi, "Pickup and delivery of partial loads with 'soft' time windows," *Amer. J. Math. Manage. Sci.*, vol. 6, nos. 3–4, pp. 369–398, 1986.

[15] Y. Dumas, J. Desrosiers, and F. Soumis, "The pickup and delivery problem with time windows," *Eur. J. Oper. Res.*, vol. 54, no. 1, pp. 7–22, 1991.

[16] J.-F. Cordeau, "A Branch-and-Cut algorithm for the Dial-a-Ride problem," *Oper. Res.*, vol. 54, no. 3, pp. 573–586, Jun. 2006.

[17] J.-J. Jaw, A. R. Odoni, H. N. Psaraftis, and N. H. M. Wilson, "A heuristic algorithm for the multi-vehicle advance request dial-a-ride problem with time windows," *Transp. Res. B, Methodol.*, vol. 20, no. 3, pp. 243–257, Jun. 1986.

[18] I. Ioachim, J. Desrosiers, Y. Dumas, M. M. Solomon, and D. Villeneuve, "A request clustering algorithm for Door-to-Door handicapped transportation," *Transp. Sci.*, vol. 29, no. 1, pp. 63–78, Feb. 1995.

[19] J.-F. Cordeau and G. Laporte, "A tabu search heuristic for the static multi-vehicle dial-a-ride problem," *Transp. Res. B, Methodol.*, vol. 37, no. 6, pp. 579–594, Jul. 2003.

[20] S. N. Parragh, K. F. Doerner, and R. F. Hartl, "Variable neighborhood search for the dial-a-ride problem," *Comput. Oper. Res.*, vol. 37, no. 6, pp. 1129–1138, Jun. 2010.

[21] R. Masson, F. Lehuédé, and O. Péton, "The Dial-A-Ride problem with transfers," *Comput. Oper. Res.*, vol. 41, pp. 12–23, Jan. 2014.

[22] K. Braekers, A. Caris, and G. K. Janssens, "Exact and meta-heuristic approach for a general heterogeneous dial-a-ride problem with multiple depots," *Transp. Res. B, Methodol.*, vol. 67, pp. 166–186, Sep. 2014.

[23] M. Chassaing, C. Duhamel, and P. Lacomme, "An ELS-based approach with dynamic probabilities management in local search for the Dial-A-Ride problem," *Eng. Appl. Artif. Intell.*, vol. 48, pp. 119–133, Feb. 2016.

[24] M. A. Masmoudi, K. Braekers, M. Masmoudi, and A. Dammak, "A hybrid genetic algorithm for the heterogeneous Dial-A-Ride problem," *Comput. Oper. Res.*, vol. 81, pp. 1–13, May 2017.

[25] R. M. Jorgensen, J. Larsen, and K. B. Bergvinsdottir, "Solving the Dial-a-Ride problem using genetic algorithms," *J. Oper. Res. Soc.*, vol. 58, no. 10, pp. 1321–1331, Oct. 2007.

[26] M. Diana and M. M. Dessouky, "A new regret insertion heuristic for solving large-scale dial-a-ride problems with time windows," *Transp. Res. B, Methodol.*, vol. 38, no. 6, pp. 539–557, Jul. 2004.

[27] Q. Lu and M. M. Dessouky, "A new insertion-based construction heuristic for solving the pickup and delivery problem with time windows," *Eur. J. Oper. Res.*, vol. 175, no. 2, pp. 672–687, Dec. 2006.

[28] S. N. Parragh and V. Schmid, "Hybrid column generation and large neighborhood search for the dial-a-ride problem," *Comput. Oper. Res.*, vol. 40, no. 1, pp. 490–497, Jan. 2013.

[29] C. Bongiovanni, M. Kaspi, and N. Geroliminis, "The electric autonomous dial-a-ride problem," *Transp. Res. B, Methodol.*, vol. 122, pp. 436–456, Apr. 2019.

[30] J. Paquette, J.-F. Cordeau, and G. Laporte, "Quality of service in dial-a-ride operations," *Comput. Ind. Eng.*, vol. 56, no. 4, pp. 1721–1734, May 2009.

[31] D. Kirchler and R. Wolfler Calvo, "A granular tabu search algorithm for the Dial-a-Ride problem," *Transp. Res. B, Methodol.*, vol. 56, pp. 120–135, Oct. 2013.

[32] *Uber's New Wait-Time Fee 'Fair'*. Accessed: Nov. 2, 2019. [Online]. Available: https://www.todayonline.com/singapore/ubers-new-wait-time-fee-fair-commuters-say

[33] *Waiting Time for Taxis Still Biggest Bugbear: Study*. Accessed: Nov. 2, 2019. [Online]. Available: https://www.straitstimes.com/singapore/transport/waiting-time-for-taxis-still-biggest-bugbear-study

[34] *New Grabshare Service With Slightly Cheaper Fares in Return for 5 Minutes' Wait Time*. Accessed: Nov. 2, 2019. [Online]. Available: https://www.straitstimes.com/singapore/transport/new-grabshare-service- with-slightly-cheaper-fares-in-return-for-five-minutes

[35] A. Janiak, W. A. Janiak, and M. Lichtenstein, "Tabu search on GPU," *J. UCS*, vol. 14, no. 14, pp. 2416–2426, 2008.

[36] T. V. Luong, L. Loukil, N. Melab, and E.-G. Talbi, "A GPU-based iterated tabu search for solving the quadratic 3-dimensional assignment problem," in *Proc. ACS/IEEE Int. Conf. Comput. Syst. Appl. (AICCSA)*, May 2010, pp. 1–8.

[37] J. Nickolls, I. Buck, and M. Garland, "Scalable parallel programming," in *Proc. IEEE Hot Chips 20 Symp. (HCS)*, Aug. 2008, pp. 40–53.

[38] M. Czapiński and S. Barnes, "Tabu search with two approaches to parallel flowshop evaluation on CUDA platform," *J. Parallel Distrib. Comput.*, vol. 71, no. 6, pp. 802–811, Jun. 2011.

[39] J. Szymon and Ż. Dominik, "Solving multi-criteria vehicle routing problem by parallel tabu search on GPU," *Procedia Comput. Sci.*, vol. 18, pp. 2529–2532, 2013.

[40] I. Coelho, P. Munhoz, L. Ochi, M. Souza, C. Bentes, and R. Farias, "An integrated CPU–GPU heuristic inspired on variable neighbourhood search for the single vehicle routing problem with deliveries and selective pickups," *Int. J. Prod. Res.*, vol. 54, no. 4, pp. 945–962, 2016.

[41] I. Chakroun, M. Mezmaz, N. Melab, and A. Bendjoudi, "Reducing thread divergence in a GPU-accelerated branch-and-bound algorithm," *Concurrency Comput., Pract. Exper.*, vol. 25, no. 8, pp. 1121–1136, Jun. 2013.

[42] N. Melab, T. Van Luong, K. Boufaras, and E.-G. Talbi, "ParadisEO-MO-GPU: A framework for parallel GPU-based local search metaheuristics," in *Proc. 15th Annu. Conf. Genetic Evol. Comput. Conf. (GECCO)*, 2013, pp. 1189–1196.

[43] J.-F. Cordeau, M. Gendreau, G. Laporte, J.-Y. Potvin, and F. Semet, "A guide to vehicle routing heuristics," *J. Oper. Res. Soc.*, vol. 53, no. 5, pp. 512–522, May 2002.

[44] F. W. Glover and G. A. Kochenberger, Eds. *Handbook of Metaheuristics*, vol. 57. Boston, MA, USA: Springer, 2006.

[45] J. Silberholz and B. Golden, "Comparison of metaheuristics," in *Handbook of Metaheuristics*. Boston, MA, USA: Springer, 2010, pp. 625–640.

[46] R. R. Pandi, S. G. Ho, S. C. Nagavarapu, T. Tripathy, and J. Dauwels, "GPU-accelerated tabu search algorithm for Dial-A-Ride problem," in *Proc. 21st Int. Conf. Intell. Transp. Syst. (ITSC)*, Nov. 2018, pp. 2519–2524.

[47] M. W. P. Savelsbergh, "The vehicle routing problem with time windows: Minimizing route duration," *ORSA J. Comput.*, vol. 4, no. 2, pp. 146–154, May 1992.

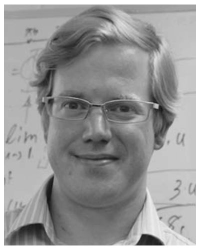
[48] W. P. Nanry and J. Wesley Barnes, "Solving the pickup and delivery problem with time windows using reactive tabu search," *Transp. Res. B, Methodol.*, vol. 34, no. 2, pp. 107–121, Feb. 2000.

[49] F. Glover, "Tabu search—Part I," *ORSA J. Comput.*, vol. 1, no. 3, pp. 190–206, 1989.

Ramesh Ramasamy Pandi received the B.E. degree in electronics and communication from Anna University, India, in 2014, and the M.Sc. degree in signal processing from Nanyang Technological University (NTU), Singapore, in 2016, where he is currently pursuing the Ph.D. degree with the School of Electrical and Electronic Engineering. He has been awarded the NTU Research Scholarship for the doctoral program. He works at the intersection of combinatorial optimization, computational intelligence, and high-performance computing. His current research interests are in operations research techniques to improve real-time decision making capability in vehicle routing with applications to intelligent transportation systems. He is a recipient of the prestigious Nanyang award, the Mr. and Mrs. Kwok Chin Yan Award, and the Best Paper Award.

Song Guang Ho was born in Kuala Lumpur, Malaysia. He received the B.Eng. degree in electrical and electronics engineering from Universiti Malaysia Sabah, Malaysia, in 2016, and the M.Sc. degree in computer control and automation from Nanyang Technological University (NTU), Singapore, in 2017. He was a Research Associate with the ST Engineering-NTU Robotics Corporate Laboratory, NTU, until 2018. Since 2018, he has been a Research Engineer with ST Engineering Land Systems, Singapore. His main areas of interest are intelligent transportation systems, operations research, combinatorial optimization, and computational algorithms.

Sarat Chandra Nagavarapu (Member, IEEE) received the B.Sc. and M.Sc. degrees in electronics from Andhra University, Visakhapatnam, India, in 2006 and 2008, respectively, the M.Tech. degree in information technology from the International Institute of Information Technology (IIIT) Bangalore, Bengaluru, India, in 2010, and the Ph.D. degree in systems and control engineering from IIT Bombay, Mumbai, India, in 2016. He is currently a Research Fellow of the Satellite Research Centre (SaRC), Nanyang Technological University (NTU), Singapore. Prior to joining this position, he was a Research Fellow of the Energy Research Institute @ NTU (ERI@N) and the ST Engineering-NTU Robotics Corporate Laboratory, NTU. His research interests include multirobot systems, vehicle routing algorithms, and designing cooperative exploration and mapping strategies for autonomous agents.



Justin Dauwels (Senior Member, IEEE) received the Ph.D. degree in electrical engineering from the Swiss Federal Institute of Technology (ETHZ), Zurich, in 2005. He was a Post-Doctoral Fellow of the RIKEN Brain Science Institute from 2006 to 2007 and a Research Scientist at the Massachusetts Institute of Technology from 2008 to 2010. He is currently an Associate Professor with the School of Electrical and Electronic Engineering, Nanyang Technological University (NTU), Singapore. He also serves as the Deputy Director of the ST Engineering-NTU Robotics Corporate Laboratory, which comprises researchers developing novel autonomous systems for airport operations and transportation. His research on intelligent transportation systems has been featured by the BBC, Straits Times, Lianhe Zaobao, Channel 5, and numerous technology websites. His research interests are in data analytics with applications to intelligent transportation systems, autonomous systems, and the analysis of human behavior and physiology. He was a JSPS Post-Doctoral Fellow in 2007, a BAEF Fellow in 2008, a Henri-Benedictus Fellow of the King Baudouin Foundation in 2008, and a JSPS Invited Fellow from 2010 to 2011.