Scalable FastMDP for Pre-departure Airspace Reservation and Strategic De-conflict

Joshua R. Bertram   Joseph Zambreno
Iowa State University
Ames, IA 50011
{bertram1, zambreno}@iastate.edu

Peng Wei
George Washington University
Washington, DC 20052
pwei@gwu.edu

Pre-departure flight plan scheduling for Urban Air Mobility (UAM) and cargo delivery drones will require on-demand scheduling of large numbers of aircraft. We demonstrate an algorithm known as FastMDP-GPU that performs first-come-first-served pre-departure flight plan scheduling where conflict free flight plans are generated on demand. We demonstrate a parallelized implementation of the algorithm on a Graphics Processor Unit (GPU) and show the level of performance and scaling that can be achieved. Our results show that on commodity GPU hardware we can perform flight plan scheduling against 2000-3000 known flight plans and with server-class hardware the performance can be higher. Our results show promise for implementing a large scale UAM scheduler capable of performing on-demand flight scheduling that would be suitable for both centralized or distributed flight planning systems.

I. Introduction

Urban Air Mobility (UAM) is an envisioned air transportation concept in which intelligent aircraft will safely and efficiently transport passengers and cargo within urban areas by rising above traffic congestion on the ground. Aircraft companies such as Boeing, Airbus, Bell, Embraer, Joby, Kitty Hawk, Pipistrel, and Volocopter are building and testing electric vertical take-off and landing (eVTOL) aircraft to ensure UAM becomes an integral part of daily life [1]. Meanwhile, in order to make UAM operations scalable, new airspace management concepts for highly dynamic and dense air traffic are being studied by NASA, FAA, Uber, Airbus, etc [2][3][4]. Furthermore, NASA's UAM Grand Challenge was announced to engage the UAM community and promote public confidence through a series of system level safety and integration scenarios [5].

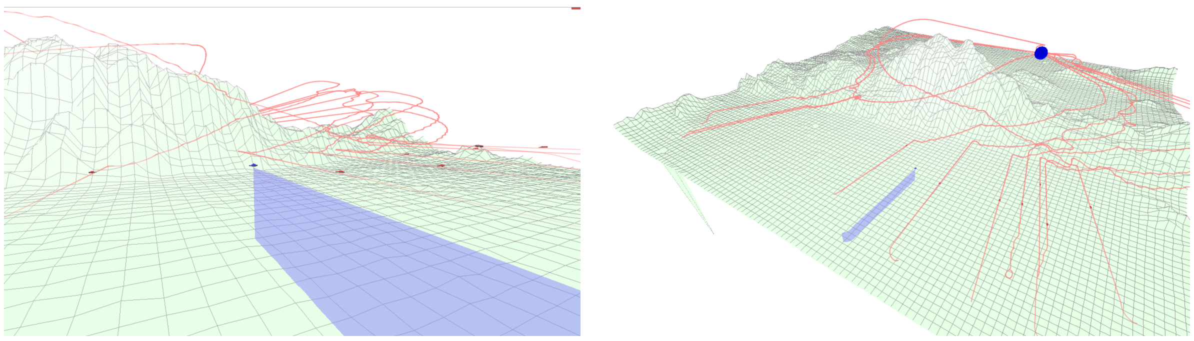


Fig. 1 Agent (blue aircraft and blue flight track) flying through an airspace with known flight plans of other aircraft (red aircraft and red trajectories).

A major challenge to scale the UAM operations and accommodate high-density urban air traffic is strategic de-conflict for eVTOL flights. “Strategic" here refers to pre-departure, whose procedure has to be completed while the aircraft is still on the ground, instead of tactical real-time conflict resolution or collision avoidance. The essence of “strategic de-conflict" is how to reserve airspace volume in a safe and efficient manner [6][7]. In this work, we focus on efficient and scalable algorithms for pre-departure airspace reservation in free flight strategic de-conflict. We assume all the airspace volume in a certain region (except the obstacles or restricted area) can be occupied by airspace users, instead of a structured airspace with routes and waypoints.

Our contribution of this paper is that this is the first scalable algorithm for pre-departure airspace reservation and strategic de-conflict to support free flight UAM operations. We respect the airspace user fairness by following a first-come-first served (FCFS) policy. The algorithm offers great flexibility for implementation. It can be run by either a fleet dispatch center (e.g. Uber), a personal user (personal owned eVTOL pilot), or by the airspace manager (e.g. the FAA). The algorithm can handle both by-scheduled UAM operations (processing flight plans or airspace reservation requests in batches) or by-request air taxi like operations (processing flight plans or airspace reservation requests one by one). Our framework exploits both route and take-off time to achieve strategic de-conflict quickly and efficiently. The algorithm provides a recommended flight plan, which contains not only the recommended route (a series of dynamic airspace volume reservations), but also the recommended take-off time from several candidates.

The approach used in this paper formulates the problem as a Markov Decision Process (MDP) and then uses an algorithm known as FastMDP to quickly and efficiently solve the MDP to generate a conflict free trajectory. A request is submitted to the system containing an aircraft identifier, a source location, and a destination location and the system returns a conflict free flight path to the user. When a conflict free flight path is generated, it is stored by the system as an “accepted” flight plan in a database of accepted flight plans. Any future requests will consider both terrain and all previously accepted flight plans when generating new conflict free flight plans. Requests can consist of a single aircraft, or can consist of batches of aircraft. The implementation of the algorithm described in this paper takes advantage of parallelization inherent in the problem to create a massively parallel implementation which scales to thousands of aircraft on commodity Graphics Processing Units (GPUs). This implementation is used to study the level of scalability that can be achieved by the algorithm on different classes of hardware. While the algorithm is capable of running on both server class GPU hardware and embedded class GPU boards suitable for low size-weight and power applications, the focus of this paper is on GPU hardware that is too large and power hungry for putting on board a typical aircraft.

The paper is structured as follows: Section II contains related work, Section III contains background material on Markov Decision Processes (MDPs) and the FastMDP algorithm, Section IV describes the method used in this paper, Section V describes the experimental setup, Section VI describes the experimental results, and the paper closes with Section VII where the conclusion and future work are discussed.

II. Related Work

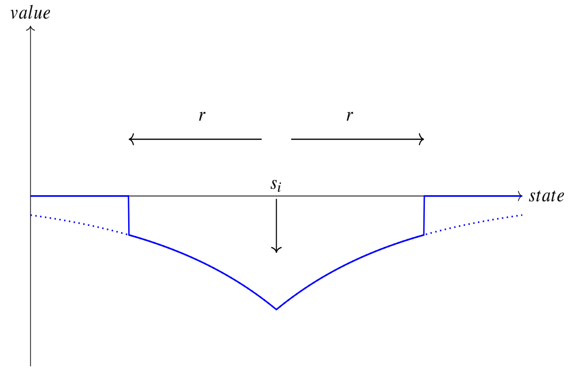
There have been many important contributions to the topic of guidance algorithms with collision avoidance capability for small unmanned aerial aircraft.

Many papers apply different techniques to manage aircraft from a centralized controller by formulating the problem as an optimal control problem. These methods can be based on semidefinite programming [8], nonlinear programming [9, 10], mixed integer linear programming [11–14], mixed integer quadratic programming [15], sequential convex programming [16, 17], second-order cone programming [18], evolutionary techniques [19, 20], and particle swarm optimization [21]. Besides formulating this problem using optimal control framework, methods such as visibility graph [22] Voronoi diagrams [23], and A^* search [24, 25] can also handle the path planning problem for aircraft. These methods work well for 2D or 3D waypoint planning, but fail to scale when aircraft dynamics are considered. To address this issue, sample-based planning algorithms have been proposed, such as probabilistic roadmaps (PRM) [26], rapidly-exploring random trees (RRT) [27], and RRT* [28].

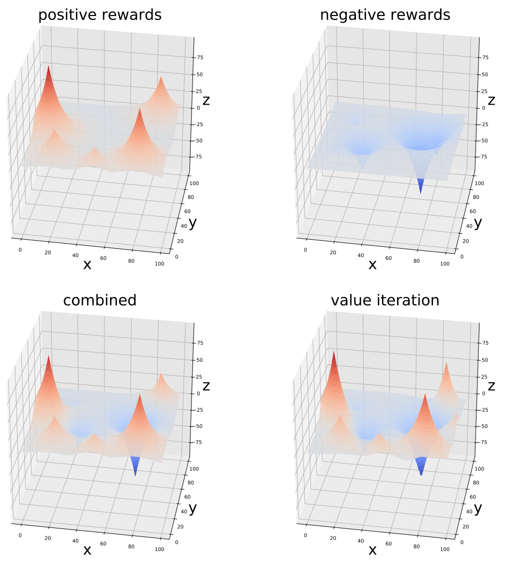
Model Predictive Control [29, 30] can be used to solve collision avoidance problem but the computation load is relatively high. Potential fields [31, 32] are computationally fast, but in general they provide no guarantees of collision avoidance. Machine learning and reinforcement learning based algorithms [33–36] have promising performance, but typically require a long offline training time. Monte Carlo Tree Search (MCTS) algorithm [37] does not need time to train before the flight and can provide an "anytime" result depending on the available computation time, but the aircraft can only adopt several discretized actions at each time step. A geometric approach [38–41] can be also applied for the collision avoidance problem and the computation time only grows linearly as the number of aircraft increases. DAIDALUS (Detect and Avoid Alerting Logic for Unmanned Systems) [42] is another geometric approach developed by NASA. The core logic of DAIDALUS consists of: (1) definition of self-separation threshold (SST) and well-clear violation volume (WCV), (2) algorithms for determining if there exists potential conflict between aircraft pairs within a given lookahead time, and (3) maneuver guidance and alerting logic. The drawback of these geometric approaches is that it can not look ahead for more than one step (it only pays attention to the current action and does not take account of the effect of subsequent actions) and the outcome can be locally optimal in the view of the global trajectory.

There are many well known methods for solving MDPs including value iteration and policy iteration, which are iterative methods based on the dynamic programming approach proposed by Bellman [43]. These algorithms use a table-based approach to represent the state-action space exactly and iteratively converge to the optimal policy π^* and corresponding value function V^*. These table-based methods have a well known disadvantage that they quickly become intractable. As the number of states and actions increases in number or dimension, the number of entries in the table increases exponentially. Many real world problems quickly exhaust the resources of even high performing computers due to the well known 'curse of dimensionality' [43]. Many attempts have been made to allow MDPs to scale to larger problems. Factored MDPs [44, 45] attempt to alleviate the problem of state space explosion by identifying subsets of the MDP that can be broken into smaller problems. Approximation methods under the general umbrella of Approximate Dynamic Programming have been used as a compromise to obtain reasonable approximations of the underlying true value function in cases where the state-action space (or the transition matrix T) is too large to represent with traditional exact methods, which are summarized by the [46, 47]. Notably, linear function approximation methods such as GradientTD methods [48–50], statistical evaluation methods such as Monte Carlo Tree Search [51], and non-linear function approximation methods such as TDC with non-linear function approximation [52] and DQN [53] are good examples of some of the approaches taken using approximation.

Traffic Management Initiatives (TMIs), including Ground Delay Program, Airspace Flow Program and Collaborative Trajectory Options Program, are set of tools air traffic managers use to balance air traffic demand with airspace capacity [54–56]. The essence of these programs is to apply ground delay or assign longer route to flights who would otherwise experience more expensive and unsafe air delay. Within the area of pre-departure flight planning, [6] defines a moving dynamic geofence around an aircraft during its flight. The dynamic geofence represents a safety margin around the aircraft which is sufficiently large to guarantee safe separation. The dynamic geofence is formulated as a convex polyhedra and intersection tests between polyhedra are used to detect conflicts with other flight plans. Network optimization is used to find feasible paths between a grid of waypoints representing the possible paths the aircraft can take when navigating between points in an urban area. Experiments show an airspace utilization improvement of 70-80% over a reserved corridor around the flight plan for the duration of the flight. This concept is expanded upon in [7] where the dynamic geofence is used, but the solution is formulated as a two-level linear programming problem. The first level of the problem is a discretized version of the problem which resolves scheduling conflicts using integer programming. The second level performs speed profile smoothing using linear programming on the discretized solution from the first level. While the method provides a global optimum, no simulation or numerical results are provided for the runtime for sample problem sizes. In [57] NASA explores and extension to AutoResolver for UAVs to model realistic air traffic management scenarios in the Dallas-Fort Worth metroplex, studying loss of separation and resolutions in a simplified structured airspace, and shows AutoResolver can effectively introduce ground delays and alter fixed flight paths along the structured airspace routes to avoid conflicts. Performance is shown in terms of the impacts to flight schedules, though performance time to run the algorithm itself was not reported. In [58], NASA explores the use of Mission Planner which performs pre-departure flight planning for UAVs / UAM in a first-come-first-served manner. Mission Planner models both the network routing and trajectory generation problems. For a given set of goods or people that need moved through the network, Mission Planner takes into account suitable aircraft availability, vertiport capacity, and generates a set of flights that will satisfy the demand. It then builds candidate flight paths and iteratively uses a set of resolution strategies to resolve conflicts or constraint violations that are discovered through the duration of each flight, resulting in a viable, conflict free flight path. Performance of the algorithm is examined in terms of scheduling a random set of realistic flights over a 3 hour window, where the number of flights was randomly sampled to be from 1000 to 10000 distributed over the 3 hour time window. A study is performed on the effectiveness of each resolution strategy. No results are provided for the run time of the algorithm itself.



(a) A risk well showing exponential decay of a negative reward out to a fixed radius beyond which the negative penalty is truncated.



(b) FastMDP solving positive and negative rewards, combining the results, and comparing to the solution produced by the value iteration algorithm traditionally used to solve MDPs.

Fig. 2 FastMDP solves MDP using peaks that represent positive and negative rewards

III. Background

Markov Decision Processes (MDPs) are a framework for decision making with broad applications to finance, robotics, operations research and many other domains [59]. MDPs are formulated as the tuple (S, A, R, T) where $s_t \in S$ is the state at a given time t, $a_t \in A$ is the action taken by the agent at time t as a result of the decision process, $r_t = R(s_t, a_t)$ is the reward received by the agent as a result of taking the action a_t from s_t, where $R(s_t, a_t)$ is known as the *reward function*, and T is the transition function that describes the evolution of the system. The dynamics of the environment are described by the transition function $T(s_t, a, s_{t+1})$ and capture the probability $P(s_{t+1}|s_t, a_t)$ of transitioning to a state s_{t+1} given the action a_t taken from state s_t. A policy π can be defined that maps each state $s \in S$ to an action $a \in A$. From a given policy $\pi \in \Pi$ a value function $V^{\pi}(s)$ can be computed that computes the expected return that will be obtained within the environment by following the policy π. We use the *infinite horizon discounted reward formulation* where a parameter $\gamma \in (0, 1)$ is defined which is applied at each step to determine return. A small value of γ favors short term reward versus long term reward, whereas a large value of γ near 1.0 favors long term reward versus short term reward.

The solution of an MDP is termed the optimal policy π^*, which defines the optimal action $a^* \in A$ that can be taken from each state $s \in S$ to maximize the expected return. From this optimal policy π^* the optimal value function $V^*(s)$ can be computed which describes the maximum expected value that can be obtained from each state $s \in S$. And from the optimal value function $V^*(s)$, the optimal policy π^* can also easily be recovered.

We refer to the path taken through the state space as a result of following the optimal policy as the *optimal trajectory*. We define the UAV planning problem as finding the optimal trajectory through the space such that the UAV maximizes its future expected reward. From any starting state, by following the optimal policy π^*, we are guarantees to also follow the optimal trajectory. MDPs are interesting because their solution simultaneously provides the optimal action a^* to perform from every state and can be viewed as analogous to a vector field in a continuous space.

A challenge with traditional MDP solution methods is that they often take a great deal of time to solve due to the iterative nature that is required to solve them. One also finds that as the number of states or actions grows, the amount of time or memory required to solve the MDP grows exponentially leading to issues of intractability. This is somewhat mitigated by the use of approximation methods which lead to tractable solution methods for MDPs which, while iterative, lead to solutions within for many interesting problems on time scales ranging from seconds to days depending on the particular problem. MDPs are typically not suitable for real-time applications, though there are some on-line methods which allow MDPs to be solved without an explicit pre-training phase.

The FastMDP algorithm proposed in [60] represents a radical departure from the traditional approach to solving MDPs. FastMDP solves a certain useful subclass of MDPs much more quickly than traditional methods by taking advantage of structure within the value function. FastMDP relies on the observation that positive and negative rewards in an MDP can be described as exponentially decaying *peaks* in the value function which can be combined in a particular way to reconstruct the value function.

In [60], a method is described to combine the positive and negative peaks together such that they closely approximate the value function produced by solving a MDP using traditional methods, as shown in Figure 2b. For UAV collision avoidance problems in [60–62] positive rewards are modeled as exponentially decaying peaks while negative rewards are modelled as *risk wells* which decay exponentially out to a fixed radius, where they are then truncated. Risk wells capture the idea that if a penalty is far enough away from an agent's current position it can be safely ignored, while also encoding that the closer the agent is to a negative reward the riskier it is to be near that reward. As described in [60], the risk well formulation also has the advantage that it can be processed using the same efficient algorithm that is used to process positive rewards. This leads to a very efficient way to model UAV collision avoidance problems that has been successfully demonstrated to solve interesting, practical problems.

Peaks are described by the tuple $\mathbf{P}_i = \{r_i, \gamma_i, \hat{\mathbf{p}}_i, R_i\}$ where the reward r_i is the scalar value of the positive or negative reward scalar that is placed in the environment, the discount factor $\gamma_i \in (0, 1)$ is used to determine discounted future reward, the position in 3D space is represented by $\hat{\mathbf{p}}_i$, and the radius of the peak is represented by R_i. Positive peaks share the same discount factor of γ_i which is also the discount factor for the overall MDP. Negative rewards may have independent values of γ_i and are not tied to the overall MDP's discount factor. (See [60] for a detailed explanation.)

The psuedocode for the FastMDP algorithm from [61] is shown in Algorithm 1 for context in understanding the overall flow before the optimizations are described. In summary, positive and negative peaks are constructed from the positive and negative rewards in the environment which are provided as inputs to the FastMDP algorithm. Positive peaks are created from the positive rewards. All negative rewards result in negative peaks which are constructed in *standard positive form*, which means that negative rewards are treated temporarily by the algorithm as if they were positive, are used in generating the solution, and are reverted back to negative values before the algorithm generates its final answer. [60] describes this process in much more detail along with the theory behind it.

With the positive and negative peaks created, the algorithm then performs forward projection of the aircraft's state for each possible action that can be taken $a \in A$ over a planning window that (in this implementation) is 10 time steps into the future, resulting in a set of points $\mathbf{\Delta_{10}}$. The value $\mathbf{V}^*[s_t]$ is computed at each point $s_t \in \mathbf{\Delta_{10}}$. The action a^* which leads to the most valuable action is then computed and is then selected for the next time step. This process then causes the algorithm to take the optimal action that it can perform at each time step.

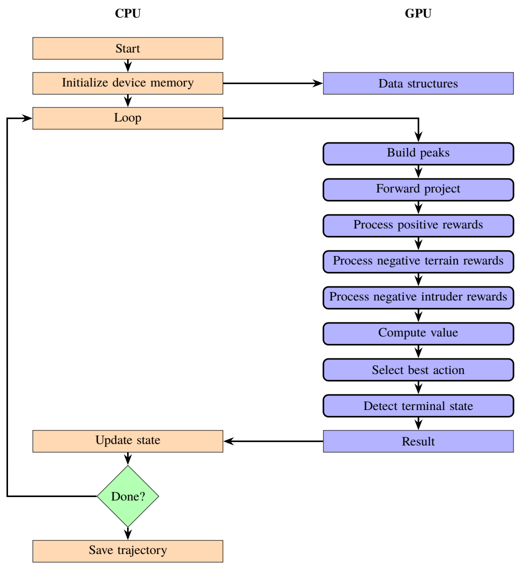
IV. Method

The method used in this paper extends the FastMDP algorithm described in [61, 62] by reimplementing it for a Graphics Processing Unit (GPU) to take advantage of parallelism inherent in the algorithm which we term *FastMDP-GPU*. The FastMDP algorithm CPU-based implementations previously published in [60–62] are efficient in that they achieve $O(|R| \times |A|)$ performance where $|R|$ is the number of positive and negative rewards and $|A|$ is the number of actions. For a given problem where the number of actions $|A|$ is fixed, the variation in run-time performance can then be considered as linear in the number of rewards $O(|R|)$. However, as the number of rewards scales to large numbers, even this $O(|R|)$ scaling is not enough to achieve real-time performance above 50-100 aircraft on a typical CPU, depending on the problem, because the number of actions $|A|$ for realistic problems is still large (even though it may be fixed for a given problem.) In order to scale to large numbers of aircraft, we must recognize that each independent action can be computed separately which is a form of parallelism that can be exploited. Furthermore, additional insights in this paper show that there is additional parallelism inherent in the processing of the rewards that can yield additional performance gains.

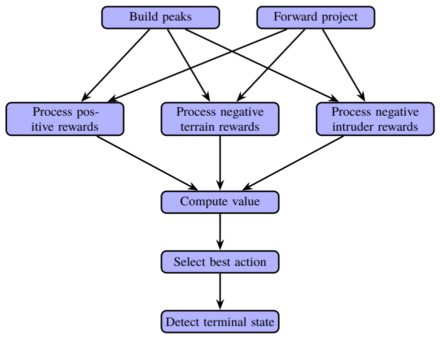
These insights on the parallel nature of the algorithm lead to the GPU based approach described in this paper, which

Algorithm 1 CPU based FastMDP algorithm from [61]

```
1: procedure DistributedUam(aircraftState, worldState)
2:     $\mathbf{S_t} \leftarrow \mathbf{S}_0$ // randomized initial aircraft states
3:     $\mathbf{A} \leftarrow$ aircraft actions (precomputed)
4:     $\mathbf{L} \leftarrow$ aircraft limits (precomputed)
5:     $\mathbf{S}_{t+1} \leftarrow$ allocated space
6:     while aircraft remain do
7:         for each aircraft do
8:             $s_t \leftarrow \mathbf{S}_t[aircraft]$
9:             // Build peaks from rewards in the environment
10:            $\mathbf{P}^+ \leftarrow$build pos rewards
11:            $\mathbf{P}^- \leftarrow$build neg rewards in Standard Positive Form
12:            $\mathbf{P}^* \leftarrow$build neg rewards for terrain in Standard Positive Form
13:            // Perform forward projection
14:            $\mathbf{\Delta_1} \leftarrow fwdProject(s_t, \mathbf{A}, \mathbf{L}, 0.1\ s)$
15:            $\mathbf{\Delta_{10}} \leftarrow fwdProject(s_t, \mathbf{A}, \mathbf{L}, 1.0\ s)$
16:            // Compute the value at each reachable state
17:            $\mathbf{V}^* \leftarrow$ allocate space for each reachable state
18:            for $s_j \in \mathbf{\Delta_{10}}$ do
19:                // First for positive peaks
20:                for $p_i \in \mathbf{P}^+$ do
21:                    // distance
22:                    $d_p \leftarrow \left\| s_j - \mathbf{location}(p_i) \right\|_2$
23:                    $r_p \leftarrow \mathbf{reward}(p_i)$
24:                    $\gamma_p \leftarrow \mathbf{discount}(p_i)$
25:                    $\mathbf{V}^+(p_i) \leftarrow |r_p| \cdot \gamma_p^{d_p}$
26:                $V^+_{max} \leftarrow \max_{p_i} \mathbf{V}^+$
27:                // Next for negative peaks (in Standard Positive Form) including terrain
28:                for $n_i \in \{\mathbf{P}^-, \mathbf{P}^*\}$ do
29:                    //distance
30:                    $d_n \leftarrow \left\| s_j - \mathbf{location}(n_i) \right\|_2$
31:                    $r_n \leftarrow \mathbf{reward}(n_i)$
32:                    $\gamma_n \leftarrow \mathbf{discount}(n_i)$
33:                    // within radius
34:                    $\rho_n \leftarrow negDist_i < \mathbf{radius}(n_i)$
35:                    $\mathbf{V}^-(p_i) \leftarrow int(\rho_n) \cdot |r_n| \cdot \gamma_n^{d_n}$
36:                $V^-_{max} \leftarrow \max_{p_i} \mathbf{V}^-$
37:                // Hard deck penalty
38:                if $\mathbf{altitude}(s_t) < penaltyAlt$ then
39:                    $V_{deck} \leftarrow 1000 - \mathbf{altitude}(s_t)$
40:                else
41:                    $V_{deck} \leftarrow 0$
42:                $\mathbf{V}^*[s_t] \leftarrow V^+_{max} - V^-_{max} - V_{deck}$
43:            // Identify the most valuable action
44:            $a^* \leftarrow \arg\max_s(\mathbf{V}^*)$
45:            // For illustration, the corresponding value
46:            $maxValue \leftarrow \mathbf{V}^*[a^*]$
47:            // And the next state when taking the action
48:            $\mathbf{s_{t+1}} \leftarrow \mathbf{\Delta_1}[a^*]$
49:            $\mathbf{S}_{t+1}[aircraft] \leftarrow \mathbf{s_{t+1}}$
50:        // Now that all aircraft have selected an action, apply it
51:        $\mathbf{S_t} \leftarrow \mathbf{S}_{t+1}$
```



(a) Flowchart of the operations performed on the CPU versus the GPU.



(b) Data dependencies between the kernels. Note that the kernels are scheduled serially by the CUDA library in the current implementation.

Fig. 3 The algorithm is broken into multiple kernels which are scheduled in a pipeline on the GPU.

allows many parts of the problem to be computed in parallel, resulting in an overall reduction in computation time required for the same set of inputs. We use this GPU based method to study the level of scaling that can be achieved by the algorithm by applying it to a pre-departure flight planning problem where conflict free flight path trajectories are computed in a first-come-first-served order.

The overall FastMDP-GPU algorithm used in this paper is shown in Figure 3a. The Central Processing Unit (CPU) code is responsible for initialization and coordination of the processing flow, and the GPU side is responsible for performing the parallel computations. The GPU code is implemented using NVIDIA's CUDA library and the GPU code blocks are referred to as *kernels*. Each kernel is designed to be a simple block of code with as few loops and branches as possible. When a kernel is launched by the CPU, on the GPU side many instances of the code within the kernel are launched in parallel *threads*. The threads are scheduled by the CUDA library to run on the available GPU cores in the hardware and run in batches until all threads have completed. The GPU is designed to handle large numbers of threads with very little overhead for switching between threads (unlike CPUs). It is not unusual for a kernel launch to run millions of threads on the GPU.

In CUDA, threads are organized into blocks, which are in turn organized into a grid. The blocks within a grid can be indexed by 1D, 2D, or 3D indices. Likewise, the threads within a block can also be indexed by 1D, 2D, or 3D indices. In the implementation used in this paper, the kernels shown in Figure 3a use different indexing schemes as needed in order to maximize parallelism and are summarized in Table 1. The kernels are arranged in a pipeline with the sequence defined by the CPU. The kernels themselves run serially, but the threads of each kernel run in parallel with each other. Individual thread scheduling is managed by the CUDA library and GPU.

Each kernel consumes one or more data structures defined in the GPU card's memory and outputs results into one or more other data structures in the GPU memory. Copying memory between the CPU and GPU is minimized and is primarily performed during initialization to set up the GPU state before the algorithm runs. At the end of each cycle, a minimal amount of state information is copied from the GPU memory to the CPU memory so that the CPU software is aware of the current state of the simulation. The major data structures used as inputs and outputs of kernels are described in Table 2.

Each kernel is now described in detail.

Table 1 Major data structures in GPU memory as inputs or outputs of each kernel, where N indicates the number of aircraft being simulated in the batch, A indicates the number of actions that can be taken, R_p indicates the number of positive rewards, R_t indicates the number of negative terrain rewards, R_i indicates the number of negative intruder rewards, and W indicates the time step window of 10. Data structures which are multi-field structures are indicated including the number of fields in the dimensionality (e.g., $N \times 6$ for a structure with 6 fields.) Sample values shown for $N = 1, A = 1350, R_p = 1, R_t = 50, R_i = 2000$ which are typical values used for a batch size of 1, an action space resulting in 1350 possible actions at each time step, terrain modelled with 50 negative rewards, and 2000 intruders.

Symbol	Purpose	Data Type	Dimensionality	Sample size (bytes)
$\mathbf{P}^+$	Peaks formed from positive rewards (eg., the goal)	64-bit float	$R_p \times 6$	$1 \times 6 \times 8 = 48$
$\mathbf{P}^-$	Peaks formed from negative rewards from other aircraft in the batch	64-bit float	$5 \times (N-1) \times 6$	$5 \times 0 \times 6 \times 8 = 0$
$\mathbf{P}^\mathbf{I}$	Peaks formed from negative rewards from intruders	64-bit float	$5 \times R_i \times 6$	$5 \times 2000 \times 6 \times 8 = 480,000$
$\mathbf{P}^\mathbf{T}$	Peaks formed from negative rewards from terrain	64-bit float	$R_t \times 6$	$50 \times 6 \times 8 = 2,400$
$\mathbf{\Delta_{10}}$	States resulting from forward projection	64-bit float	$N \times A \times W \times 12$	$1 \times 1350 \times 10 \times 12 \times 8 = 1,296,000$
$\mathbf{V}^+$	Value contributed at states due to contributions from positive rewards from $\mathbf{P}^+$	64-bit float	$N \times A \times W$	$1 \times 1350 \times 10 \times 8 = 108,000$
$\mathbf{V}^-$	Value contributed at states due to contributions from negative rewards from $\mathbf{P}^-$	64-bit float	$N \times A \times W$	$108,000$
$\mathbf{V}^\mathbf{I}$	Value contributed at states due to contributions from negative rewards from $\mathbf{P}^\mathbf{I}$	64-bit float	$N \times A \times W$	$108,000$
$\mathbf{V}^\mathbf{T}$	Value contributed at states due to contributions from negative rewards from $\mathbf{P}^\mathbf{T}$	64-bit float	$N \times A \times W$	$108,000$
$\mathbf{V}$	Value computed from all positive and negative contributions (primarily for debug and visualization)	64-bit float	$N \times A \times W$	$108,000$
$\mathbf{V}^*$	Value computed from all positive and negative contributions	64-bit float	$N \times W$	$1 \times 10 \times 8 = 80$
$\mathbf{A}^*$	Selected action for each aircraft	64-bit float	N	$1 \times 8 = 8$

Table 2 Kernel indexing schemes used for each kernel, where N indicates the number of aircraft being simulated in the batch, A indicates the number of actions that can be taken, R_p indicates the number of positive rewards, R_t indicates the number of negative terrain rewards, and R_i indicates the number of negative intruder rewards. Sample values shown for $N = 1, A = 1350, R_p = 1, R_t = 50, R_i = 2000$ which are typical values used for a batch size of 1, an action space resulting in 1350 possible actions at each time step, terrain modelled with 50 negative rewards, and 2000 intruders.

Name	Dimensionality	Indexes	Sample number of threads
Build peaks	1D	N	1
Forward project	2D	$N \times A$	1350
Process positive rewards	3D	$N \times A \times R_p$	1350
Process negative rewards	3D	$5 \times N \times A \times (N-1)$	0
Process negative terrain rewards	3D	$N \times A \times R_t$	67,500
Process negative intruder rewards	3D	$5 \times N \times A \times R_i$	13,500,000
Compute value	2D	$N \times A$	1350
Select best action	1D	N	1
Determine terminal state	3D	$N \times N$	1

A. Kernel: Build peaks

Peaks are built as described in Table 3. In this implementation, each aircraft is assigned a single goal location which represents a vertiport or other landing site. A single positive peak is created to model the goal. For each intruder, multiple risk wells (negative rewards) are defined at different points along the intruder's current trajectory as defined by the position and the linear velocity of the intruder.

The inputs of the algorithm are the current state of the aircraft in the batch and the intruders. The outputs of the kernel are the positive peaks $\mathbf{P}^{+}$, negative peaks $\mathbf{P}^{-}$, and negative peaks for intruders $\mathbf{P}^{\mathbf{I}}$.

Note that terrain features are also modelled with risk wells, but these peaks $\mathbf{P}^{\mathbf{T}}$ are defined statically at load time and transferred to GPU memory during the algorithm initialization phase. Also note that this kernel is called once for each aircraft in the batch (see Table 1). While it contains loops, profiling has shown that the kernel contributes negligible overhead and needs no further optimization. The logic in this kernel is more suitable for operation on CPU, but it is implemented as a kernel primarily to avoid unnecessary copying to and from CPU and GPU memory.

Algorithm 2 Build Peaks Kernel

```
1: procedure BUILD PEAKS($i_{ac}$)
2:     // $i_{ac}$: Aircraft index
3:     // $i_a$: Action index

4:     // Build peaks per Table 3
5:     // Build positive for the goal
6:     $\mathbf{P}^{+} \leftarrow \{r_g, \gamma = 0.9, \hat{\mathbf{p}}_g, \infty\}$ // goal
7:     // Build negative rewards for each other aircraft in the batch
8:     for $ac \in batch$ do
9:         $\mathbf{P}^{-}{}_{ac} \leftarrow \{r_{ac}, \gamma_{ac}, \hat{\mathbf{p}}_{ac}, R_{ac}\}$ // aircraft
10:    // Build negative rewards for each intruder
11:    for $ac \in intruders$ do
12:        $\mathbf{P}^{\mathbf{I}}{}_{ac} \leftarrow \{r_{ac}, \gamma_{ac}, \hat{\mathbf{p}}_{ac}, R_{ac}\}$ // aircraft
```

Table 3 Peaks created in the environment. For each aircraft in the environment, multiple negative peaks are placed along its trajectory. Terrain features in this implementation are negative rewards placed manually to overlay the terrain. Each aircraft's goal is also manually selected and represents a vertiport in the environment. Here $\hat{\mathbf{p}}$ represents the position of an aircraft and $\hat{\mathbf{v}}$ represents the velocity of the aircraft.

For each intruder or other aircraft in the batch:

Magnitude	Decay factor	Location	Radius	Timesteps	Comment
-1000	.97	$\hat{\mathbf{p}} + \hat{\mathbf{v}}t$	$300 + 10t$	$\forall t \in \{-5, 0, 5, 10, 15\}$	Collision avoidance, 5 rewards

For each terrain feature:

Magnitude	Decay factor	Location	Radius	Timesteps	Comment
-1000	.99	manually placed	manually selected	N/A	Terrain avoidance

For aircraft's goal:

Magnitude	Decay factor	Location	Radius	Timesteps	Comment
200	.999	manually placed	∞	N/A	Vertiport attraction

B. Kernel: Forward project

Forward projection here refers to using models of the aircraft dynamics to compute the future state of the aircraft based on an assumed action for a fixed duration of time. The aircraft dynamics and actions used in this paper are the same as those used in [61]. The forward projection used here is considered a module that can be replaced with another physics model or forward projection method. Additionally, multiple physics models that model different aircraft types could also be implemented and used to simulate different aircraft dynamics within the same simulation.

The input of this kernel is the current state of all aircraft in the batch and the set of all possible actions that an aircraft can take from the current state. One thread is created for each aircraft in the batch and each action that can be taken from the current state for $N \times A$ total threads (see Table 1).

The output of this kernel is the future state of all aircraft in the batch for each possible action for each time step in the lookahead window.

The time window forward projection is performed over is denoted with W and represents the number of simulation time steps of duration dt to perform. In this implementation, $W = 10$ and each time step is $dt = 0.1$ seconds. The time window could be increased or decreased and is selected so that the forward projected actions provide a significant enough spread in the state space for the algorithm to detect a difference in value between states. The key here is that the forward projection needs to be far enough away for the agent to react to the truncated boundary of risk wells. Less maneuverable aircraft models will require a larger forward projection window W.

Note here that nothing precludes the time step dt from being variable. For simplicity, in this implementation the time step is fixed, but if an adaptive time step dt or time window W were desired, this is achievable without loss of generality. Likewise, at different time scales, different dynamics models could be used which are appropriate for the timescale if an adaptive fidelity approach were desired.

Note also that if the dynamics model is such that the points in time can be computed independently from each other, then the kernel could be further parallelized, but we do not assume this to always be the case.

Algorithm 3 Forward Projection Kernel

```
1: procedure FORWARD PROJECTION($i_{ac}$, $i_a$)
2:     // $i_{ac}$: Aircraft index
3:     // $i_a$: Action index

4:     // Perform forward projection of aircraft dynamics given action
5:     for $t \in \{1, \cdots, W\}$ do
6:         $\mathbf{\Delta_{10}}[i_{ac}, i_a, t] \leftarrow$ step physics computations forward in time
```

C. Kernel: Process positive rewards

For each of the future states computed by forward projection, the value contributed by positive peaks is computed and saved in an output array.

The input of this kernel is the forward projected state for each action for each time step dt in the forward projection window W. One thread is created for each aircraft in the batch for each action that can be taken from the current state for each positive peak for $N \times A \times R_p$ total threads (see Table 1).

The output of this kernel is the value at each state that is contributed by each positive peak.

Algorithm 4 Positive Rewards Kernel

```
1: procedure POSITIVE REWARDS($i_{ac}$, $i_a$, $i_p$)
2:     // $i_{ac}$: Aircraft index
3:     // $i_a$: Action index
4:     // $i_p$: Peak index

5:     for $t \in \{1, \cdots, W\}$ do
6:         // Get projected state for this time step
7:         $s \leftarrow \mathbf{\Delta_{10}}[i_{ac}, i_a, t]$
8:         // Get peak
9:         $p_i \leftarrow \mathbf{P}^+[i_p]$
10:        // Compute distance between state and peak
11:        $d_p \leftarrow \|s - \mathbf{location}(p_i)\|_2$
12:        // Extract reward magnitude for the peak from data structure
13:        $r_p \leftarrow \mathbf{reward}(p_i)$
14:        // Extract discount factor for the peak from data structure
15:        $\gamma_p \leftarrow \mathbf{discount}(p_i)$
16:        // Compute the value with respect to the peak
17:        $V \leftarrow |r_p| \cdot \gamma_p^{d_p}$
18:        // Save max value (atomic operation)
19:        $\mathbf{V}^+[i_{ac}, i_a, t] \leftarrow \max\left(\mathbf{V}^+[i_{ac}, i_a, t], V\right)$
```

D. Kernel: Process negative rewards

For each of the future states computed by forward projection, the value contributed by negative peaks is computed and saved in an output array.

The input of this kernel is the forward projected state for each action for each time step dt in the forward projection window W. One thread is created for each aircraft in the batch for each action that can be taken from the current state for each negative peak for $N \times A \times R_n$ total threads (see Table 1).

The output of this kernel is the value at each state that is contributed by each negative peak.

Algorithm 5 Negative Rewards Kernel

```
1: procedure NEGATIVE REWARDS($i_{ac}, i_a, i_p$)
2:     // $i_{ac}$: Aircraft index
3:     // $i_a$: Action index
4:     // $i_p$: Peak index

5:     for $t \in \{1, \cdots, W\}$ do
6:         // Get projected state for this time step
7:         $s \leftarrow \mathbf{\Delta_{10}}[i_{ac}, i_a, t]$
8:         // Get peak
9:         $p_i \leftarrow \mathbf{P}^{-}[i_p]$
10:        // Compute distance between state and peak
11:        $d \leftarrow \|s - \mathbf{location}(p_i)\|_2$
12:        // Extract reward magnitude for the peak from data structure
13:        $r \leftarrow \mathbf{reward}(p_i)$
14:        // Extract discount factor for the peak from data structure
15:        $\gamma \leftarrow \mathbf{discount}(p_i)$
16:        // Extract radius for the peak from data structure
17:        $R \leftarrow \mathbf{radius}(p_i)$
18:        // Compute whether we are inside the radius of the peak, result is a 1 if true or a 0 if false.
19:        $in \leftarrow d < R$
20:        // Compute the value with respect to the peak
21:        $V \leftarrow in \cdot |r| \cdot \gamma^d$
22:        // Save max value (atomic operation)
23:        $\mathbf{V}^{-}[i_{ac}, i_a, t] \leftarrow \max(\mathbf{V}^{-}[i_{ac}, i_a, t], V)$
```

E. Kernel: Process negative terrain rewards

For each of the future states computed by forward projection, the value contributed by terrain peaks is computed and saved in an output array.

The input of this kernel is the forward projected state for each action for each time step dt in the forward projection window W. One thread is created for each aircraft in the batch for each action that can be taken from the current state for each terrain peak for $N \times A \times R_t$ total threads (see Table 1).

The output of this kernel is the value at each state that is contributed by each terrain peak.

Algorithm 6 Terrain Rewards Kernel

```
1: procedure TERRAIN REWARDS($i_{ac}, i_a, i_p$)
2:     // $i_{ac}$: Aircraft index
3:     // $i_a$: Action index
4:     // $i_p$: Peak index

5:     for $t \in \{1, \cdots, W\}$ do
6:         // Get projected state for this time step
7:         $s \leftarrow \mathbf{\Delta_{10}}[i_{ac}, i_a, t]$
8:         // Get peak
9:         $p_i \leftarrow \mathbf{P^T}[i_p]$
10:        // Compute distance between state and peak
11:        $d \leftarrow \|s - \mathbf{location}(p_i)\|_2$
12:        // Extract reward magnitude for the peak from data structure
13:        $r \leftarrow \mathbf{reward}(p_i)$
14:        // Extract discount factor for the peak from data structure
15:        $\gamma \leftarrow \mathbf{discount}(p_i)$
16:        // Extract radius for the peak from data structure
17:        $R \leftarrow \mathbf{radius}(p_i)$
18:        // Compute whether we are inside the radius of the peak, result is a 1 if true or a 0 if false.
19:        $in \leftarrow d < R$
20:        // Compute the value with respect to the peak
21:        $V \leftarrow in \cdot |r| \cdot \gamma^d$
22:        // Save max value (atomic operation)
23:        $\mathbf{V^T}[i_{ac}, i_a, t] \leftarrow \max\left(\mathbf{V^T}[i_{ac}, i_a, t], V\right)$
```

F. Kernel: Process negative intruder rewards

For each of the future states computed by forward projection, the value contributed by intruder peaks is computed and saved in an output array.

The input of this kernel is the forward projected state for each action for each time step dt in the forward projection window W. One thread is created for each aircraft in the batch for each action that can be taken from the current state for each intruder peak for $N \times A \times R_i$ total threads (see Table 1).

The output of this kernel is the value at each state that is contributed by each intruder peak.

Algorithm 7 Intruder Rewards Kernel

```
1: procedure INTRUDER REWARDS($i_{ac}, i_a, i_p$)
2:     // $i_{ac}$: Aircraft index
3:     // $i_a$: Action index
4:     // $i_p$: Peak index

5:     for $t \in \{1, \cdots, W\}$ do
6:         // Get projected state for this time step
7:         $s \leftarrow \mathbf{\Delta_{10}}[i_{ac}, i_a, t]$
8:         // Get peak
9:         $p_i \leftarrow \mathbf{P^I}[i_p]$
10:        // Compute distance between state and peak
11:        $d \leftarrow \|s - \mathbf{location}(p_i)\|_2$
12:        // Extract reward magnitude for the peak from data structure
13:        $r \leftarrow \mathbf{reward}(p_i)$
14:        // Extract discount factor for the peak from data structure
15:        $\gamma \leftarrow \mathbf{discount}(p_i)$
16:        // Extract radius for the peak from data structure
17:        $R \leftarrow \mathbf{radius}(p_i)$
18:        // Compute whether we are inside the radius of the peak, result is a 1 if true or a 0 if false.
19:        $in \leftarrow d < R$
20:        // Compute the value with respect to the peak
21:        $V \leftarrow in \cdot |r| \cdot \gamma^d$
22:        // Save max value (atomic operation)
23:        $\mathbf{V^I}[i_{ac}, i_a, t] \leftarrow \max\left(\mathbf{V^I}[i_{ac}, i_a, t], V\right)$
```

G. Kernel: Compute value

In this pipeline stage, the results of all positive and all negative rewards are combined into a single value for each state computed by forward projection. One thread is created for each aircraft in the batch for each action that can be taken from the current state for $N \times A$ total threads (see Table 1).

Algorithm 8 Compute Value Kernel

```
1: procedure COMPUTE VALUE($i_{ac}, i_a$)
2:     // $i_{ac}$: Aircraft index
3:     // $i_a$: Action index

4:     // Initialize the total value
5:     $V_{max} \leftarrow 0$
6:     // Compute the maximum value that this action resulted in along its trajectory
7:     for $t \in \{1, \cdots, W\}$ do
8:         // Get the maximum positve, negative, terrain, and intruder values at this time step.
9:         $V^+_{max} \leftarrow \mathbf{V}^+[i_{ac}, i_a, t]$
10:        $V^-_{max} \leftarrow \mathbf{V}^-[i_{ac}, i_a, t]$
11:        $V^-_{max} \leftarrow \mathbf{V^T}[i_{ac}, i_a, t]$
12:        $V^T_{max} \leftarrow \mathbf{V^I}[i_{ac}, i_a, t]$
13:        // Compute an altitude penalty if we go below a hard deck minimum altitude
14:        $V_{alt} \leftarrow$ apply penalty if less than hard deck
15:        // Compute the value from all the components
16:        $V \leftarrow V^+_{max} - \max\left(V^-_{max}, V^T_{max}, V^I_{max}\right) - V_{alt}$
17:        $V_{max} \leftarrow \max\left(V_{max}, V\right)$
18:    // Save off the maximum value this this action achieved
19:    $\mathbf{V}^*[i_{ac}, i_a] \leftarrow V_{max}$
```

H. Kernel: Select best action

Here the value of each possible action is computed and the action with maximum value is identified and selected. One thread is created for each aircraft in the batch for N total threads (see Table 1).

Algorithm 9 Select Action Kernel

```
1: procedure SELECT ACTION($i_{ac}$)
2:     // $i_{ac}$: Aircraft index

3:     // Identify the most valuable action
4:     $a^* \leftarrow \max_{a \in A} \mathbf{V}^*[i_{ac}, a]$
5:     $\mathbf{A}^*[i_{ac}] = a^*$
```

I. Kernel: Determine terminal state

This function monitors all aircraft in the simulation to detect collisions, Near Mid-Air Collisions (NMACs), collisions with terrain, and aircraft that have successfully reached their goals. Selected actions are also applied to the internal simulation state which effectively advances simulation time by dt. This code operates on the GPU side to avoid having to transfer memory to the CPU. A summary of the results of this function and of the current simulation state are passed to the CPU side to determine if simulation should continue or end. As this is just simple accounting and collision testing, the code is omitted.

V. Experimental Setup

To demonstrate the effectiveness of the algorithm in solving a real-world problem, we apply the algorithm to a pre-departure flight planning (PDFP) problem in which an aircraft must determine a flight plan before takeoff which has no conflicts with any other aircraft's previously accepted flight plan. In this setup, an aircraft submits a starting and ending location to the PDFP system, and the PDFP system returns a trajectory to the aircraft which is known to be conflict free. The PDFP system maintains a database of terrain and of *accepted* flight plans, which are the result of previous successful requests for flight plans from other aircraft who have utilized the first-come-first-served PDFP request system. The PDFP system guarantees that the result of a PDFP request is conflict free with respect to terrain and with all accepted flight plans. When a new flight plan is generated by the PDFP system, it is automatically stored in the database of accepted flight plans and is then used for any new future requests.

This could represent a centralized planning system where all aircraft submit to a central service provider, such as NASA's UTM. This could also represent a distributed case where an operator of a fleet or individual aircraft build a flight plan using a published set of flight plans. In this case, multiple operators could build flight plans simultaneously that are conflict free with respect to the published set of flight plans. However, there would need to be some mechanism which resolves conflicting flight plans that result from distributed planners who inadvertently plans which conflict with each other but are conflict free with respect to the published flight plans. Such a mechanism is outside the scope of this paper.

The implementation for this paper is focused on determining the level of scalability that can be achieved by the algorithm and does not put any effort into the database design. The database used here is a simple table in memory which stores a list of previously accepted conflict free flight plans. This database is loaded from a file and a mechanism is available to add an accepted flight plan to this file. In a larger scale implementation for a production environment, this could be implemented by a production quality database system such as SQL server or equivalent. Likewise, in this implementation, it is naively assumed that all simulation time steps should take place at a 0.1 second increment. In a more sophisticated implementation, a larger time step could be taken in regions where it is safe to do so (e.g., away from other aircraft and terrain). This would have the effect of being able to complete a flight plan in many fewer iterations of the algorithm and would lead to a higher throughput of the pre-departure flight planning system, but would not impact the level of scalability of the algorithm itself being studied in this paper.

While the implementation in this paper plans for a single source and destination, it can easily be extended to plan along a route using known points (e.g., NAVAIDs) as waypoints. In this way, multiple routes could be explored in parallel by allowing the algorithm to generate several candidate schedules which are returned to the requester. The requester could then evaluate different attributes of the flight plan, such as fuel usage, risk profile, etc and select a preferred route. Likewise by performing parallel scheduling at different departure times, varying routes can be examined. For example, it may be that leaving 30 minutes earlier avoids known traffic congestion and leads to lower fuel cost.

Each request causes the algorithm in this paper to be invoked with one or more aircraft. All previously accepted flight plans are treated as intruders and are made available to the algorithm. The algorithm is allowed to execute until it reaches its goal or a collision occurs during simulation. Collisions with terrain or with other aircraft are tracked and reported. Only a trajectory without collisions is accepted, otherwise an error is reported.

For batch sizes greater than one, there are two ways in which to operate this algorithm. One possibility is to invoke the algorithm with multiple aircraft being co-simulated together with the same set of accepted flights. In this mode, the aircraft will be aware of each other and will route around each other and all accepted flight plans. This is necessary if the aircraft within the batch might possibly intersect each other's flight plans (e.g., when they are flying through an overlapping flight volume.) In cases where requests can be segregated into non-overlapping flight volumes (e.g. separate sections of a metropolitan area, separate sectors of airspace, etc), then independent parallel instances of the algorithm can be run independent of each other. For a successful large scale implementation of this algorithm, a strategy should be employed to break requests into manageable numbers of overlapping flights, and to co-simulate small numbers of flights together in batches allowing independent batches to run in parallel. An implementation of this level of parallelism is left to future work.

While in this paper, all accepted flight plans are generated by successively running the algorithm with a sequence of source and destinations, in principle the flight plans could also be generated by some external source and imported into this algorithm. The flight plans could be in the form of trajectory points as is currently done in this implementation, or could be in some higher level summary form such as line segments, Bezier splines, or some other more efficient representation.

VI. Results

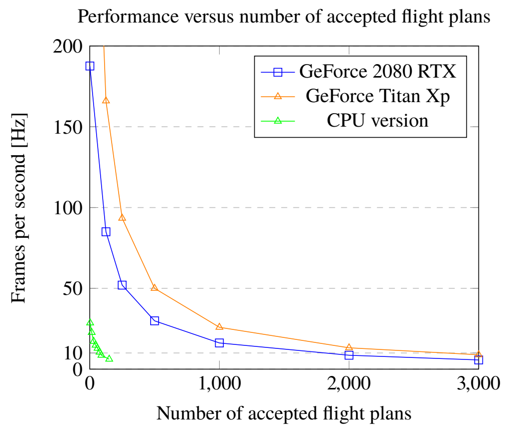
Figure 4a shows results for the algorithm's performance as the number of accepted flight plans increases. The GeForce 2080 RTX used for this test has 8GB of RAM and tests were performed with NVIDA driver version 441.66 running CUDA 10.2 on Windows 10. The GeForce Titan XP has 12GB of RAM, used driver 440.60 running CUDA 10.2 on Ubuntu 18.04.

Performance results as the number of accepted flight plans varies are shown in Figure 4a. Any value above 10 Hz represents running faster than real-time. This crossover occurs near 2,000 accepted flight plans on the GeForce 2080 RTX, which represents approximately 10,000 reward peaks that are processed by the algorithm. On the GeForce Titan XP, the crossover occurs at about 3,000 accepted flight plans (15,000 reward peaks). On the CPU version of this algorithm, this same crossover happens at around 75 flight plans. These results clearly show the benefit that the GPU version of the algorithm is able to produce to reach higher scaling.

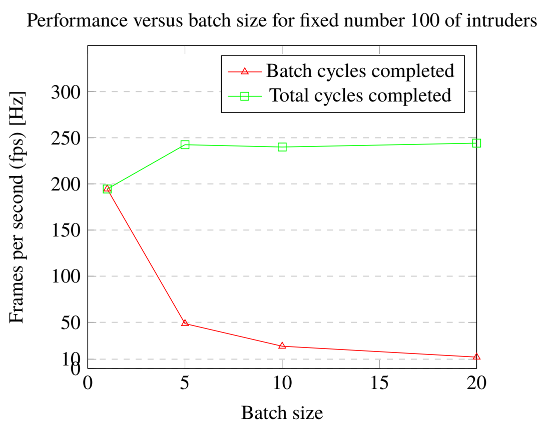
Performance results as the batch size varies are shown in Figure 4b. While this demonstrates that the frame rate for processing the batch decreases as the number of aircraft in the batch increase, it also shows that if the total cycles processed (the x-coordinate multiplied by the y-coordinate) is considered, it reflects that the GPU becomes saturated and can perform a certain amount of work (certain number of computations) per second. Once the GPU becomes saturated (at $x = 5$ on the graph), it can no longer keep up with the amount of new work being requested. This saturation point will occur at a different level depending on the power of the GPU.

A. The impact of departure delay on en route flight time

To demonstrate the feasibility of examining multiple departure schedules, we show the results of varying the departure time on the resulting flight plan. For illustration purposes, we define the cost function $t_{\text{flight}} = t_a - t_d$ as each flight's en route time, where t_d is the departure time and t_a is the arrival time. In general other cost function could be applied (e.g., fuel or electricity usage based off thrust, control actions, passenger comfort, risk profile, etc.). We first generate a set of conflict free flight plans to form a congested airspace by allowing the algorithm to schedule a number of flight plans in succession all departing at time $t_d = 0$ so that we have a set of known flight plans which are designed to generate traffic



(a) Performance results as the number of accepted flight plans varies. GPU performance greatly exceeds the CPU performance from [61].



(b) Performance results as the batch size varies.

Fig. 4 Performance results of the algorithm.

congestion we can schedule around. We then perform a set of i candidate ground delay requests for each flight which delay the departure time by some time $t_i = c * i$ where c is an arbitrary delay constant such as 25 seconds. So the new departure time will be $t'_d = t_d + t_i$. We then show the resulting en route flight time from running the algorithm with each candidate departure time in Table 4. In this experiment, waiting approximately 200 seconds is a good option to avoid all congestion and obtain the shortest flight time which in turn would result in lower fuel cost.

Table 4 Impact of departure delay on en route flight time

Departure delay (s)	En route flight time (s)
0	579
25	545
50	536
75	588
100	535
125	539
150	538
175	537
200	518
300	518
400	518
500	518
600	518

VII. Conclusion

This paper presents FastMDP-GPU, an approach for performing pre-departure flight planning that is efficient and scales to a large number of aircraft using a highly parallelized GPU-based approach. This research is the first work that efficiently computes flyable pre-departure trajectories and strategically deconflicts with thousands of other planned trajectories. This approach has the potential to perform pre-departure flight planning scheduling for dense UAM and other congested environments which require quick replanning.

For future work, further optimization can be performed on the algorithm to obtain higher levels of utilization of the GPU. Additionally, multi-GPU systems can be utilized to perform concurrent processing on multiple GPUs. Likewise, moving processing into the cloud or a cluster so that multiple GPUs on separate systems are utilized could also improve performance. Alternatively, given the limitations of a particular GPU another strategy could be to segment the problem by geographical area. Geographic Information Systems (GIS) databases often employ queries which operate over an area which are often implemented as range queries. Range query algorithms could be used to restrict the number of trajectories that need to be considered to a volume relevant to the expected flight path.

References

[1] Uber Elevate, "Fast-Forwarding to a Future of On-Demand Urban Air Transportation," 2016.

[2] Gipson, L., "NASA Embraces Urban Air Mobility, Calls for Market Study," `https://www.nasa.gov/aero/nasa-embraces-urban-air-mobility`, Nov. 2017. Acessed Nov 20, 2018.

[3] Uber Elevate, "Airspace Management at Scale: Dynamic Skylane Networks," 2nd Annual Uber Elevate Summit, 2018.

[4] "Urban Air Mobility," `http://publicaffairs.airbus.com/default/public-affairs/int/en/our-topics/Urban-Air-Mobility.html`, 2018. Accessed: 2018-08-13.

[5] NASA, "NASA's UAM Grand Challenge," `https://www.nasa.gov/uamgc`, 2019. Acessed Aug 25, 2019.

[6] Zhu, G., and Wei, P., "Low-Altitude UAS Traffic Coordination with Dynamic Geofencing," *16th AIAA Aviation Technology, Integration, and Operations Conference*, 2016, p. 3453.

[7] Zhu, G., and Wei, P., "Pre-Departure Planning for Urban Air Mobility Flights with Dynamic Airspace Reservation," *AIAA Aviation 2019 Forum*, 2019, p. 3519.

[8] Frazzoli, E., Mao, Z.-H., Oh, J.-H., and Feron, E., "Resolution of conflicts involving many aircraft via semidefinite programming," *Journal of Guidance, Control, and Dynamics*, Vol. 24, No. 1, 2001, pp. 79–86.

[9] Raghunathan, A. U., Gopal, V., Subramanian, D., Biegler, L. T., and Samad, T., "Dynamic optimization strategies for three-dimensional conflict resolution of multiple aircraft," *Journal of guidance, control, and dynamics*, Vol. 27, No. 4, 2004, pp. 586–594.

[10] Enright, P. J., and Conway, B. A., "Discrete approximations to optimal trajectories using direct transcription and nonlinear programming," *Journal of Guidance, Control, and Dynamics*, Vol. 15, No. 4, 1992, pp. 994–1002.

[11] Schouwenaars, T., De Moor, B., Feron, E., and How, J., "Mixed integer programming for multi-vehicle path planning," *Control Conference (ECC), 2001 European*, IEEE, 2001, pp. 2603–2608.

[12] Richards, A., and How, J. P., "Aircraft trajectory planning with collision avoidance using mixed integer linear programming," *American Control Conference, 2002. Proceedings of the 2002*, Vol. 3, IEEE, 2002, pp. 1936–1941.

[13] Pallottino, L., Feron, E. M., and Bicchi, A., "Conflict resolution problems for air traffic management systems solved with mixed integer programming," *IEEE transactions on intelligent transportation systems*, Vol. 3, No. 1, 2002, pp. 3–11.

[14] Vela, A., Solak, S., Singhose, W., and Clarke, J.-P., "A mixed integer program for flight-level assignment and speed control for conflict resolution," *Decision and Control, 2009 held jointly with the 2009 28th Chinese Control Conference. CDC/CCC 2009. Proceedings of the 48th IEEE Conference on*, IEEE, 2009, pp. 5219–5226.

[15] Mellinger, D., Kushleyev, A., and Kumar, V., "Mixed-integer quadratic program trajectory generation for heterogeneous quadrotor teams," *Robotics and Automation (ICRA), 2012 IEEE International Conference on*, IEEE, 2012, pp. 477–483.

[16] Augugliaro, F., Schoellig, A. P., and D'Andrea, R., "Generation of collision-free trajectories for a quadrocopter fleet: A sequential convex programming approach," *Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on*, IEEE, 2012, pp. 1917–1922.

[17] Morgan, D., Chung, S.-J., and Hadaegh, F. Y., "Model predictive control of swarms of spacecraft using sequential convex programming," *Journal of Guidance, Control, and Dynamics*, Vol. 37, No. 6, 2014, pp. 1725–1740.

[18] Acikmese, B., and Ploen, S. R., "Convex programming approach to powered descent guidance for mars landing," *Journal of Guidance, Control, and Dynamics*, Vol. 30, No. 5, 2007, pp. 1353–1366.

[19] Delahaye, D., Peyronne, C., Mongeau, M., and Puechmorel, S., "Aircraft conflict resolution by genetic algorithm and B-spline approximation," *EIWAC 2010, 2nd ENRI International Workshop on ATM/CNS*, 2010, pp. 71–78.

[20] Cobano, J. A., Conde, R., Alejo, D., and Ollero, A., "Path planning based on genetic algorithms and the monte-carlo method to avoid aerial vehicle collisions under uncertainties," *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, IEEE, 2011, pp. 4429–4434.

[21] Pontani, M., and Conway, B. A., "Particle swarm optimization applied to space trajectories," *Journal of Guidance, Control, and Dynamics*, Vol. 33, No. 5, 2010, pp. 1429–1441.

[22] Hoffmann, G., Rajnarayan, D. G., Waslander, S. L., Dostal, D., Jang, J. S., and Tomlin, C. J., "The Stanford testbed of autonomous rotorcraft for multi agent control (STARMAC)," *The 23rd Digital Avionics Systems Conference (IEEE Cat. No. 04CH37576)*, Vol. 2, IEEE, 2004, pp. 12–E.

[23] Howlet, J. K., Schulein, G., and Mansur, M. H., "A practical approach to obstacle field route planning for unmanned rotorcraft," 2004.

[24] Meng, B.-b., and Gao, X., "UAV path planning based on bidirectional sparse A* search algorithm," *Intelligent Computation Technology and Automation (ICICTA), 2010 International Conference on*, Vol. 3, IEEE, 2010, pp. 1106–1109.

[25] Xia, L., Jun, X., Manyi, C., Ming, X., and Zhike, W., "Path planning for UAV based on improved heuristic A* algorithm," *2009 9th International Conference on Electronic Measurement Instruments*, 2009, pp. 3–488–3–493.

[26] Kavraki, L., Svestka, P., and Overmars, M. H., *Probabilistic roadmaps for path planning in high-dimensional configuration spaces*, Vol. 1994, Unknown Publisher, 1994.

[27] LaValle, S. M., "Rapidly-exploring random trees: A new tool for path planning," 1998.

[28] Karaman, S., and Frazzoli, E., "Sampling-based algorithms for optimal motion planning," *The international journal of robotics research*, Vol. 30, No. 7, 2011, pp. 846–894.

[29] Shim, D. H., and Sastry, S., "An evasive maneuvering algorithm for UAVs in see-and-avoid situations," *American Control Conference, 2007. ACC'07*, IEEE, 2007, pp. 3886–3891.

[30] Shim, D. H., Kim, H. J., and Sastry, S., "Decentralized nonlinear model predictive control of multiple flying robots," *Decision and control, 2003. Proceedings. 42nd IEEE conference on*, Vol. 4, IEEE, 2003, pp. 3621–3626.

[31] Sigurd, K., and How, J., "UAV trajectory design using total field collision avoidance," *AIAA Guidance, Navigation, and Control Conference and Exhibit*, 2003, p. 5728.

[32] Langelaan, J., and Rock, S., "Towards autonomous UAV flight in forests," *AIAA Guidance, Navigation, and Control Conference and Exhibit*, 2005, p. 5870.

[33] Kahn, G., Zhang, T., Levine, S., and Abbeel, P., "Plato: Policy learning using adaptive trajectory optimization," *Robotics and Automation (ICRA), 2017 IEEE International Conference on*, IEEE, 2017, pp. 3342–3349.

[34] Zhang, T., Kahn, G., Levine, S., and Abbeel, P., "Learning deep control policies for autonomous aerial vehicles with mpc-guided policy search," *Robotics and Automation (ICRA), 2016 IEEE International Conference on*, IEEE, 2016, pp. 528–535.

[35] Ong, H. Y., and Kochenderfer, M. J., "Markov Decision Process-Based Distributed Conflict Resolution for Drone Air Traffic Management," *Journal of Guidance, Control, and Dynamics*, 2016, pp. 69–80.

[36] Chen, Y. F., Liu, M., Everett, M., and How, J. P., "Decentralized non-communicating multiagent collision avoidance with deep reinforcement learning," *Robotics and Automation (ICRA), 2017 IEEE International Conference on*, IEEE, 2017, pp. 285–292.

[37] Yang, X., and Wei, P., "Autonomous On-Demand Free Flight Operations in Urban Air Mobility using Monte Carlo Tree Search," 2018.

[38] Han, S.-C., Bang, H., and Yoo, C.-S., "Proportional navigation-based collision avoidance for UAVs," *International Journal of Control, Automation and Systems*, Vol. 7, No. 4, 2009, pp. 553–565.

[39] Park, J.-W., Oh, H.-D., and Tahk, M.-J., "UAV collision avoidance based on geometric approach," *SICE Annual Conference, 2008*, IEEE, 2008, pp. 2122–2126.

[40] Krozel, J., Peters, M., and Bilimoria, K., "A decentralized control strategy for distributed air/ground traffic separation," *AIAA Guidance, Navigation, and Control Conference and Exhibit*, 2000, p. 4062.

[41] Van Den Berg, J., Guy, S. J., Lin, M., and Manocha, D., "Reciprocal n-body collision avoidance," *Robotics research*, Springer, 2011, pp. 3–19.

[42] Muñoz, C., Narkawicz, A., Hagen, G., Upchurch, J., Dutle, A., Consiglio, M., and Chamberlain, J., "DAIDALUS: detect and avoid alerting logic for unmanned systems," 2015.

[43] Bellman, R., *Dynamic programming*, Courier Corporation, 2013.

[44] Schuurmans, D., and Patrascu, R., "Direct value-approximation for factored MDPs," *Advances in Neural Information Processing Systems*, 2002, pp. 1579–1586.

[45] Guestrin, C., Koller, D., Parr, R., and Venkataraman, S., "Efficient solution algorithms for factored MDPs," *Journal of Artificial Intelligence Research*, Vol. 19, 2003, pp. 399–468.

[46] Bertsekas, D. P., *Dynamic programming and optimal control*, Vol. 1, Athena scientific Belmont, MA, 1995.

[47] Powell, W. B., *Approximate Dynamic Programming: Solving the curses of dimensionality*, Vol. 703, John Wiley & Sons, 2007.

[48] Sutton, R. S., Maei, H. R., and Szepesvári, C., "A Convergent O(n) temporal-difference Algorithm for Off-policy Learning with Linear Function Approximation," *Advances in neural information processing systems*, 2009, pp. 1609–1616.

[49] Sutton, R. S., Maei, H. R., Precup, D., Bhatnagar, S., Silver, D., Szepesvári, C., and Wiewiora, E., "Fast gradient-descent methods for temporal-difference learning with linear function approximation," *Proceedings of the 26th Annual International Conference on Machine Learning*, ACM, 2009, pp. 993–1000.

[50] Precup, D., Sutton, R. S., and Dasgupta, S., "Off-policy temporal-difference learning with function approximation," *ICML*, 2001, pp. 417–424.

[51] Kocsis, L., and Szepesvári, C., "Bandit based monte-carlo planning," *European conference on machine learning*, Springer, 2006, pp. 282–293.

[52] Bhatnagar, S., Precup, D., Silver, D., Sutton, R. S., Maei, H. R., and Szepesvári, C., "Convergent temporal-difference learning with arbitrary smooth function approximation," *Advances in Neural Information Processing Systems*, 2009, pp. 1204–1212.

[53] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M., "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.

[54] Odoni, A. R., *The flow management problem in air traffic control*, Springer, 1987, pp. 269–288.

[55] Brennan, M., "Airspace flow programs — A fast path to deployment," *The Journal of Air Traffic Control*, Vol. 49, No. 1, 2007, pp. 51–55.

[56] Zhu, G., "Decision making under uncertainties for air traffic flow management," Ph.D. thesis, Iowa State University, 2019.

[57] Bosson, C., and Lauderdale, T. A., "Simulation Evaluations of an Autonomous Urban Air Mobility Network Management and Separation Service," *2018 Aviation Technology, Integration, and Operations Conference*, 2018, p. 3365.

[58] Guerreiro, N. M., Butler, R. W., Maddalon, J. M., and Hagen, G. E., "Mission Planner Algorithm for Urban Air Mobility–Initial Performance Characterization," *AIAA Aviation 2019 Forum*, 2019, p. 3626.

[59] Sutton, R. S., and Barto, A. G., *Reinforcement learning: An introduction*, Vol. 1, MIT press Cambridge, 1998.

[60] Bertram, J., "A new solution for Markov Decision Processes and its aerospace applications," Ph.D. thesis, Iowa State University, 2020. Masters' Thesis.

[61] Bertram, J., and Wei, P., "Distributed Computational Guidance for High-Density Urban Air Mobility with Cooperative and Non-Cooperative Collision Avoidance," *AIAA Scitech 2020 Forum*, 2020, p. 1371.

[62] Bertram, J., and Wei, P., "An Efficient Algorithm for Self-Organized Terminal Arrival in Urban Air Mobility," *AIAA Scitech 2020 Forum*, 2020, p. 0660.