Journal of Systems Architecture 154 (2024) 103239

Contents lists available at ScienceDirect

Journal of Systems Architecture

journal homepage: www.elsevier.com/locate/sysarc

GPU implementation of the Frenet Path Planner for embedded autonomous systems: A case study in the *F1tenth* scenario

Filippo Muzzini [a,*], Nicola Capodieci [a], Federico Ramanzin [b], Paolo Burgio [a]

[a] University of Modena and Reggio Emilia - Department of Physics, Informatics and Mathematics, Via Campi 213/A, Modena, 41125, Italy
[b] University of Modena and Reggio Emilia - Department of Engineering "Enzo Ferrari", Via Pietro Vivarelli 1, Modena, 41125, Italy

ARTICLE INFO

Keywords:
Planning
Autonomous vehicle
Parallel
GPU
Racing

ABSTRACT

Autonomous vehicles are increasingly utilized in safety-critical and time-sensitive settings like urban environments and competitive racing. Planning maneuvers ahead is pivotal in these scenarios, where the onboard compute platform determines the vehicle's future actions. This paper introduces an optimized implementation of the Frenet Path Planner, a renowned path planning algorithm, accelerated through GPU processing. Unlike existing methods, our approach expedites the entire algorithm, encompassing path generation and collision avoidance. We gauge the execution time of our implementation, showcasing significant enhancements over the CPU baseline (up to 22x of speedup). Furthermore, we assess the influence of different precision types (double, float, half) on trajectory accuracy, probing the balance between completion speed and computational precision. Moreover, we analyzed the impact on the execution time caused by the use of Nvidia Unified Memory and by the interference caused by other processes running on the same system. We also evaluate our implementation using the F1tenth simulator and in a real race scenario. The results position our implementation as a strong candidate for the new state-of-the-art implementation for the Frenet Path Planner algorithm.

1. Introduction

Autonomous systems (ASs) are nowadays adopted in complex and safety-critical scenarios, such as autonomous vehicles, warehouse forklifts, off-road tractors, and trucks, to cite a few. The final target of a completely autonomous vehicle is getting closer and closer, and we are currently in the transition phase of leaving the control from the human pilot to the onboard intelligence. In automotive cars, this is captured and standardized by the transition between L3/L4 SAE levels [1].

As compared to the generation of ASs categorized as L1/L2, the increased complexity and interaction with humans in the target environment require ASs to react quickly and provide timely responses within stringent timing bounds, typically a few milliseconds. The time constraint is important especially in the race context since the higher velocities shorten the necessary time to plan and execute the maneuvers [2–4]. Failure to meet these deadlines may result in functional failure and significant damage or loss of life.

In the context of autonomous vehicles, the key aspect for ensuring functional and timing correctness is the capability to efficiently process a significant amount of data within the onboard computing systems in a timely fashion. Therefore, high-performance embedded computers featuring highly parallel accelerators such as GPUs, FPGAs, or ASICs are key enabling technologies. These platforms are amenable to accelerating the complex structures of modern Deep Neural Networks, accomplishing the task of *perceiving* surrounding environment, and reconstructing a detailed view of the world. Unfortunately, most of the research effort is being spent in this direction, while other key aspects, such as *planning and control* are less explored, especially with reference to embedded systems. In this work, we partly bridge this gap. After completing the perception phase, during which the vehicle constructs a model of its surrounding environment, the vehicle must then take a decision regarding its next action. This process is typically split into two parts, a global and a local planner. The global part is often called the *mission*, in which the long-term vehicle trajectory is decided. Such a computation is performed sporadically, in an off-line manner, using well-known algorithms [5] such as A*, RRT within a few (tens of) seconds, and it is, therefore, less of our interest, as it does not involve the challenges related to on-board computing. In racing scenarios, this long-term trajectory is optimally calculated based on the reconstructed map of the known racetrack. *Local/online planners*, on the other hand, are what make these applications critical as unexpected obstacles and other dangerous situations might compromise successful navigation. Examples can be found in racing cars overtaking at high

* Corresponding author.
E-mail addresses: filippo.muzzini@unimore.it (F. Muzzini), nicola.capodieci@unimore.it (N. Capodieci), 254629@studenti.unimore.it (F. Ramanzin), paolo.burgio@unimore.it (P. Burgio).

https://doi.org/10.1016/j.sysarc.2024.103239
Received 12 April 2024; Received in revised form 26 June 2024; Accepted 12 July 2024
Available online 16 July 2024

speed, where the control algorithm response time must be adequate for such a maneuver.

This work is based on our preliminary experiments [6]. We will focus on accelerating a well-known family of strategies for local planning, namely the ones based on *Frenet* coordinate systems, which prove themselves to be highly effective, for instance, in accomplishing the task of overtaking/obstacle avoidance at high speed in racing vehicles. This algorithm is commonly used in racing competitions [7], such as Roborace [8] or Indy Autonomous Challenge [2]. The reason for this wide adoption has to be found on the *Frenet* coordinate system, as it significantly simplifies the generation of trajectories with respect to other planners, yet at the same time enables obstacle avoidance. Since it generates a set of trajectories, the execution time of this approach depends on the number of generated trajectories and typically it is performed on high-performance computing platforms. Acceleration of this algorithm will be carried out using an embedded computing platform that features a GPU accelerator. We accelerate the algorithm through the parallelization and optimization of its phases. In this manner, the math behind the algorithm and its output does not change, hence retains the property of its original formulation. The optimal acceleration of path planning and obstacle avoidance is paramount due to the high speeds (tens of km/h) at which the involved vehicles are moving. We target a representative situation, where 1/10 scale cars are deployed first in a simulator, and then in a real racetrack, and they shall perform collision avoidance of static and dynamic objects at the highest possible speed. We employ a realistic system, where computation is performed on an embedded platform with a GPU accelerator of the NVIDIA family (Jetson Xavier). The contribution of this paper is described as follows:

1. We propose a novel GPU implementation for the *Frenet Path Planner* algorithm. To the best of our knowledge, the proposed implementation is the most complete in terms of tasks executed on GPU, as we present a GPU port for the entire algorithm pipeline.
2. We test our proposed port using different precision types (*double*, *float*, *half*), aiming at understanding the impact on execution time latencies for every part of the algorithm giving more insights concerning the preliminary experiments performed in [6]. Hence, testing the precision and feasibility of the generated trajectories using the different data types, we see that half precision shows the best speedup sacrificing a little in path precision.
3. We measure the impact of the proposed implementation both in terms of execution time and precision compared to a baseline CPU implementation.
4. We studied the impact of the use of Nvidia UM (Unified Memory) and of the interference caused by other processes on the DRAM system memory.
5. We deployed our implementation on a real F1tenth vehicle, and safely tested it in the official racing simulator using a Hardware-in-the-loop setup, to devise an accurate timing profile of the application. Moreover, we show the benefits and limitations of our implementation in a real-world racing scenario.
6. We made the source code of our implementation publicly available[1] and, to the best of our knowledge, this represents the first publicly available GPU-based *Frenet Path Planner* implementation.

The paper is structured as follows: in Section 2 we explore the related work on *Planners*, *Frenet Path Planner*, and we give the reader a useful background on the GPU as accelerator. In Section 3 we describe the *Frenet Path Planner* algorithm and in Section 4 we propose our new implementation on GPU. In Section 5 we detail the test we performed using our GPU implementation and we report the results in Section 6 showing the speedup over the CPU implementation and the impact of UM and interference caused by other concurrent processes. Moreover, we show the impact of our implementation on the racing scenario on both the F1tenth simulator and a real race. We draw our conclusions in Section 7.

[1] https://github.com/HiPeRT/FrenetTenth

2. Related work

2.1. Planning

The *Planning* problem has been extensively studied in previous literature. Some approaches generate the path only considering the vehicle constraints. In [9,10], for example, a path is generated considering kinematic and dynamic constraints. The dynamic model is also considered in [11], in which a spatio-temporal lattice with vehicle feasible states is proposed. Similarly in [12,13] the kinematics quantities are optimized to find the path. In the race context, a fully reactive planner called the Follow The Gap method (FTG) [14] is often used. It focuses on avoiding any collision with any object by aiming at the center of the maximum gap/"hole" among scanned obstacles. It results in a less computationally intensive algorithm because it generates a single trajectory, which depends solely on the distance between the track walls/objects and sensor depth. However, this means that the (single) target trajectory that is selected might not be optimal, since FTG cannot implement more complex choices among multiple candidates (*e.g.*, to avoid moving objects – typically opponents). Moreover, it is highly sensitive to noise and errors in the sensor scans, resulting in a quite "fuzzy" car behavior. This might cause drifts and unexpected direction changes upon false positives/negatives sensed during the perception phase. A more complex approach using model predictive control is proposed in [15]; this approach is also used in [16,17]. In [18] the problem is formulated as an optimization problem and it is solved using the gradient descent method. In [19] the Euler spirals are used to describe paths of non-holonomic vehicles. A new curvature parametrization for this approach is presented in [20] and in [21] the model for generating velocity profiles is changed. Another approach is to generate a single path and iteratively improve it [22]. Lastly, some contributions are based on the generation of different paths to then choose the best one [23,24] according to predefined metrics. The Rapid Exploring Random Tree algorithm [25] is used in [26,27]. In the first contribution, a closed loop system is simulated for sampling a tree of trajectories, whereas in the latter the state space is explored along a given reference path.

2.2. Frenet Path Planner

The contributions cited in the previous section show evident limits in specific situations. An example of these situations is those related to the nose-to-tail traffic [28]. Moreover, the researches cited in the previous section also suffer from the inherent limits of dealing with complex formulation for curves and paths that derive when using Cartesian coordinates. For these reasons, more recent work has been focusing on methods that account for time in order to improve the algorithms for path planning but also exploiting a different coordinate system for simplifying the problem formulation. For instance, *Frenet Coordinates*, also known as *Frenet Frame*, can be exploited for these purposes. In [24] the authors use Frenet Coordinates to split the generation of lateral and longitudinal movements. Moreover, also obstacles are considered and the method attempts to generate a collision-free trajectory. More recently in [29] the authors consider dynamic objects in *Frenet Frame*. Also, our work exploits *Frenet Coordinates* for overcoming obstacles; as far as performance is concerned our method will focus on an optimized GPU implementation in which the performance analysis will also account for different settings w.r.t. data type precision.

Still referring to performance, the authors in [30] do not consider the obstacles and focus on the performance aspects of the method. In [31] the authors parallelize on the CPU the path generation. A GPU

implementation is used to accelerate the *Path Planner* in the already cited contribution [20] and in [32]. The GPU is exploited also in the acceleration of the A* algorithm [33] and in its randomized variant [34]. GPU acceleration is also exploited in the *Path Planning* of Unmanned Aerial Vehicle (UAV) in [35,36]. The previously cited contributions did not exploit *Frenet Frame*, whereas in [37] the authors accelerate on GPU only the path generation on *Frenet Coordinates* omitting to port on GPU other phases such as obstacle avoidance. Differently from [37], we also ported both obstacle avoidance and the best path selection on the GPU.

2.3. Background on GPU and CUDA

GPUs were initially designed with the goal of accelerating graphic workloads. However, the same kind of hardware can nowadays also be exploited for general-purpose computing (GPGPU). A GPU is designed as a SIMD processor (Single Instruction Multiple Data) able to process large amounts of data in a massively parallel fashion. Typically, in order to have a compute task to be executed on a GPU, it is necessary to allocate memory on GPU visible-only memory spaces, copy the input data from the host (CPU) to the GPU device memory, execute a *kernel*, which is a program specifically coded for the GPU instruction set architecture, to then copy back the results on the host memory. In order to assist the programmer in exploiting the GPU capabilities, NVIDIA released a proprietary programming model called CUDA (Compute Unified Device Architecture). CUDA provides APIs and libraries able to be accessed by high-level programming languages such as C++. In CUDA and also in other GPU APIs, kernels are dispatched through a programmer-specified launch configuration in which it is described the degree of parallelism in which the work must be computed over a three dimensional grid (X, Y, Z) of parallel threads [38]. A grid is therefore composed of blocks of threads. Threads within a single block, are able to share data and synchronize their operations. They can access a special area of memory called *Shared Memory* and they can synchronize using a special barrier (the CUDA *syncthread* directive). This barrier ensures that all the threads of the same block have performed all the instructions dispatched before such a barrier. In our path planning problem, we exploit these features to ensure that all cost components of a path are computed before reducing them to a single sum. Moreover, the CUDA programming model allows the programmer to express an added layer of parallelism through CUDA *Streams*. A CUDA *Stream* is a queue of commands (compute kernel invocations and memory movements) that must be executed sequentially in the order in which they are enqueued. As a consequence of this, a single program that manages more than one stream is able to submit commands concurrently, so that they might execute in parallel [39]. In order to synchronize the execution of commands among different *Streams*, a mechanism called CUDA *Events* can be used. In our specific problem, we exploit *Streams* to generate paths while the obstacle positions are copied to the GPU memory for the future collision avoidance. As mentioned, the data must be copied to GPU visible only memory before kernel execution. This has to be done explicitly by the programmer. Moreover, Nvidia supports an automatic mechanism to retrieve the necessary data from the host when needed. This mechanism is called Nvidia Unified Memory (UM). In this manner, explicit memory copies can be avoided: when a kernel is launched the driver automatically copies the necessary data chunks to the GPU memory. This implies that the kernel execution time can increase since there is an overhead for memory copy but this overhead can be masked by computation on already fetched data chunks. This mechanism can result in a speedup or a slowdown with respect to the use of explicit memory copies. Understanding whether or not it is convenient to perform explicit copies or to rely on unified memory is not trivial at all as it depends on the specific application [40–42]. More specifically, unified memory approaches can bring benefit to the total execution time only when the degree of compute instructions within a kernel are spread in a way to mask the memory operations from host memory. We therefore add this kind of analysis on our specific case study.

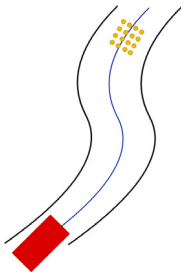


Fig. 1. Frenet Path Planner. Reference path and possible endpoints.

3. Frenet Path Planner

3.1. Overview

The path planning phase of an autonomous driving system consists in the generation of a path that the vehicles can safely follow. The path might be represented as a continuous function that links the starting point to the destination. It represents the positions that the vehicle will occupy over time. Typically such a path is discretized and it is represented as a list of path points. The path must be placed in a drivable area and this implies that the vehicle must not move to or invade spatial regions outside such an area. For instance, regions outside the drivable area could be sidewalks or pedestrian-only areas, or a wrong lane.

The path planning phase can be divided into two macro phases: *Global Planner* and *Local Planner*. The *Global Planner* generates a reference path. It is used as a guide for the *Local Planner*. The global planner does not consider obstacles on the road or vehicle constraints and does not exploit information like speed or acceleration.

The *Local Planner* generates a more detailed path considering the reference path generated by the *Global Planner*; typically this path is shorter and considers the obstacles that surround the vehicle beyond the possible maneuvers that the vehicle is able to perform.

The *Frenet Path Planner* [24] uses the *Frenet Coordinates* (Section 3.2) to compute the path and then converts it into the original *World coordinates*. Considering Fig. 1, the reference path is the blue line, which follows a road as shown in the figure. The vehicle is the red box and it must reach one of the possible endpoints (yellow dots). Initially, a set of possible paths that connect the actual vehicle position to one of the endpoints is computed. In this phase, the *Frenet Coordinates* are exploited to simplify the path generation (see Section 3.3). A cost is then assigned to each path.

The cost is a function that considers each path point and returns a scalar. A typical cost function is the cumulative jerk.

After assigning a cost to each generated path, the paths are converted into the *World coordinates* and compared with the obstacles located in the environment for collision check. Therefore, the path that features the lowest cost and that does not show any potential collision is chosen.

3.2. Frenet coordinates

Frenet Coordinates [43,44] describe the geometric properties of a curve; in our case, the curve is the reference path that overlaps the road. The *Frenet Path Planner* is based on the *Frenet Coordinates* in which s and d represent the *World coordinates* of the point (x and y) in the new *Frenet coordinates*. s is the longitudinal position along the reference path, and d is the lateral position with respect to the reference path (see Appendix A.1 for more details).

3.3. Paths generation

Considering the actual vehicle position in the *Frenet Coordinates* (s_0, d_0), a single path is a contiguous trajectory that links (s_0, d_0) to a final position (s_f, d_f).

Because one path is not sufficient for the planner, it is important to generate a set of paths that start from (s_0, d_0) and terminate in a possible endpoint (s_f, d_f). This set is generated considering: (1) the initial state $S_0 = \langle s_0, d_0, \dot{s}_0, \dot{d}_0, \ddot{d}_0 \rangle$ in which $\dot{s}_0$, $\dot{d}_0$ are the longitudinal and lateral instant velocity, $\ddot{d}_0$ is the later instant acceleration; (2) the following parameters:

- D_s: road width sampling length
- D_{max}: max road width
- D_{min}: min road width
- V_s: velocity sampling length
- V_{max}: max velocity
- V_{min}: min velocity
- V_{target}: desirable speed
- T_s: prediction time sampling length
- T_{max}: max prediction time
- T_{min}: min prediction time;

and (3) a candidate end state $S_f = \langle s_f, d_f, \dot{s}_f, \dot{d}_f, \ddot{d}_f \rangle$ in which $\dot{s}_f$ leads in the range $[V_{min}, V_{max}]$ and d_f leads in the range $[D_{min}, D_{max}]$. Moreover, it is known that this state must be reached in the time t_f that leads in the range $[T_{min}, T_{max}]$.

So we can construct the set of possible paths that start from the actual state and end in a possible end state. Each path is discretized in f points and for each of them, the position, velocity, acceleration, and jerk for both s and d are computed. Eventually, the set of possible paths PS is computed as in Algorithm 1.

Algorithm 1 Frenet Path Generation

Input: Initial state $S_0 =< s_0, d_0, \dot{s}_0, \dot{d}_0, \ddot{d}_0 >$
Output: Path set PS

for $d_f \leftarrow range(D_{min}, D_{max}, D_s)$ **do**
  for $t_f \leftarrow range(T_{min}, T_{max}, T_s)$ **do**
    $P_{tmp} = []$
    for $t \leftarrow range(0, t_f, T_s)$ **do**
      $p = pathPoint()$
      $p.t = t$
      $a_3, a_4, a_5 \leftarrow solveLinearSystem(d_f, S_0)$ // Eq. (A.4)
      $p.d = d_0 + \dot{d}_0 t + \ddot{d}_0 t^2 + a_3 t^3 + a_4 t^4 + a_5 t^5$
      $p.\dot{d} = \dot{d}_0 + 2\ddot{d}_0 t + 3a_3 t^2 + 4a_4 t^3 + 5a_5 t^4$
      $p.\ddot{d} = 2\ddot{d}_0 + 6a_3 t + 12a_4 t^2 + 20a_5 t^3$
      $p.\dddot{d} = 6a_3 + 24a_4 t + 60a_5 t^2$
      $P_{tmp} \leftarrow p$
    end for
    for $\dot{s}_f \leftarrow range(V_{min}, V_{max}, V_s)$ **do**
      $P = []$
      for $t \leftarrow range(0, t_f, T_s)$ **do**
        $p \leftarrow P_{tmp}.get(t)$
        $a_3, a_4 \leftarrow solveLinearSystem(\dot{s}_f, S_0)$ // Eq. (A.5)
        $p.s = s_0 + \dot{s}_0 t + \ddot{s}_0 t^2 + a_3 t^3 + a_4 t^4$
        $p.\dot{s} = \dot{s}_0 + 2\ddot{s}_0 t + 3a_3 t^2 + 4a_4 t^3$
        $p.\ddot{s} = 2\ddot{s}_0 + 6a_3 t + 12a_4 t^2$
        $p.\dddot{s} = 6a_3 + 24a_4 t$
        $P \leftarrow p$
      end for
      $PS \leftarrow P$
    end for
  end for
end for

Each path has an associated cost C used to determine which of the generated paths is the best.

Lastly, since the path is expressed in *Frenet Coordinates* s and d, they must be converted in the world coordinates x and y.

The interested reader can get the details behind this algorithm in Appendix A.2.

3.4. Collision check

The last part of the *Frenet Planner* is the *Collision Check*. In this part, a path P is tested for collisions against obstacles that might be present in the environment. Details are in Appendix A.3

4. Our implementation

In this section we describe our novel GPU-based implementation of *Frenet Planner*. This implementation is the same used in preliminary experiments performed in [6]. The goal is to reduce the computational time of the algorithm. The original implementation of the algorithm does not meet the necessary performance metrics in embedded boards. To understand these metrics, we need to consider racing scenarios, where the entire control pipeline must react quickly and effectively. It is crucial that the execution time of the planner aligns closely with the sensor frequency. Previous implementations using CPUs [24,29], and even those partially offloading the algorithm to the GPU [37], fail to achieve this objective. We have redesigned the original algorithm to exploit GPU capabilities by implementing all its phases in parallel. Specifically, we parallelized the loops in the original implementation and computed the path cost and world coordinates alongside the path computation (refer to Section 4.1 for details). This approach preserves the original algorithm's rationale while significantly improving completion time through careful selection of parallelization mechanisms.

We have measured the total algorithm execution time and we noticed that the *Path Generation* and *Collision Check* phases account for the majority of the execution time: about 57% for *Path Generation* and about 37% for *Collision Check*. Since both the *Path Generation* and the *Collision Check* phases perform the same work with different independent data, it is reasonable to compute them in a parallel way. For these reasons, we have implemented two CUDA kernels: one for *Path Generation* and one for *Collision Check*. The first kernel receives the system state $(s_0, d_0, \dot{s}_0, \dot{d}_0, \ddot{d}_0)$ and computes the possible paths, the second checks the paths computed by the first kernel against the obstacles resulting in a subset of paths. Moreover, we exploit CUDA *Streams* to overlap kernel execution and memory copies as in Fig. 2(a). The *Path Selection* phase is carried out on GPU employing the CuBLAS library[2], which results in reduced computational time and minimized data transfer from the GPU to the CPU. By selecting the optimal path on the GPU, only one trajectory is required to be transferred back to the CPU, in contrast to all generated trajectories.

4.1. Paths generation

The *Path Generation* phase presents several degrees of potential parallelism. Regarding the original implementation, the mathematical operations in this phase remain unchanged, but we enhanced efficiency by parallelizing path computations. Additionally, we optimized performance by integrating to the Path Generation kernel the conversion from Frenet frame to world coordinates and paths' cost assignment. These optimizations significantly reduce the execution time of this phase. Considering that the final scope of this phase is to construct the set PS (*i.e.* the set of generated paths), it is possible to compute each path

[2] https://docs.nvidia.com/cuda/cublas/

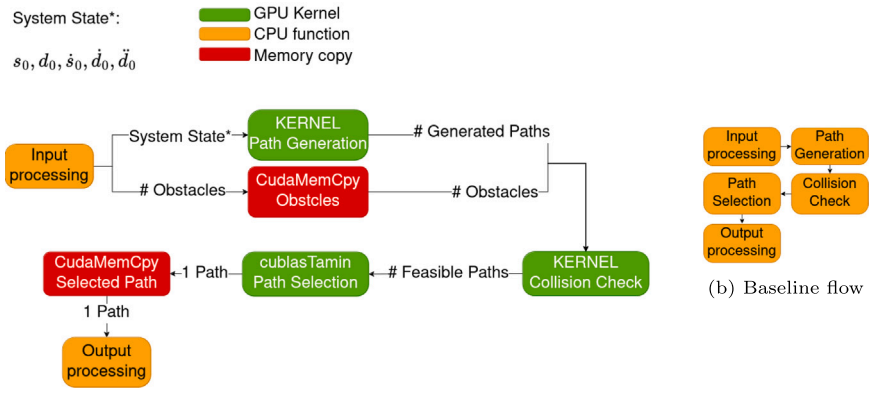


Fig. 2. Implementation flows.

$P \in PS$ independently. Moreover, each point $p \in P$ can be computed independently.

Each path $P \in PS$ is computed starting from specific values of d_f, t_f, and $\dot{s}_f$ (lateral displacement, final time, and target velocity). This corresponds to the three iterative constructs shown in Algorithm 1 (lines 1, 2, and 14). The path P is composed of different points, each of them calculated at different time t (the innermost `for` cycle in Algorithm 1 at lines 4 and 16). Since the values described above and the consequent computations are independent, we have exploited GPU parallelism to compute them concurrently.

We set up a CUDA kernel with a three-dimensional launch grid that would allow us to compute each path concurrently (see Fig. 3(a)): in x we have a block for each road width sampling d_f, in y a block for each time sampling t_f, in z a block for each velocity sampling $\dot{s}_f$. In the end, each block corresponds to a single path P and the amount of blocks present in the grid is $\frac{D_{max}-D_{min}}{D_s} \cdot \frac{T_{max}-T_{min}}{T_s} \cdot \frac{V_{max}-V_{min}}{V_s}$.

Moreover, we have mapped the computation of the points of a path to a different thread of the block associated with the path itself. By doing so, each thread computes the point of the path considering a different value of t. The kernel launch configuration is ($\frac{D_{max}-D_{min}}{D_s}$, $\frac{T_{max}-T_{min}}{T_s}$, $\frac{V_{max}-V_{min}}{V_s}$) for grid and ($\frac{T_{max}}{T_s}$,1,1) for block. We choose this configuration to exploit the *shared memory* among threads within the same block, with the aim to efficiently compute the final path cost. Since each path cost is independent of other paths we mapped each path to a different block.

Each thread computes the values d, $\dot{d}$, $\ddot{d}$, $\dddot{d}$, s, $\dot{s}$, $\ddot{s}$, $\dddot{s}$, x, and y (lateral position, velocity, acceleration and jerk; longitudinal position, velocity, acceleration and jerk; position in world coordinate) of its assigned point. Moreover, the thread computes the cost components of its associated point ($p.\ddot{s}^2$ and $p.\ddot{d}^2$) that concur to the final cost C of the path.

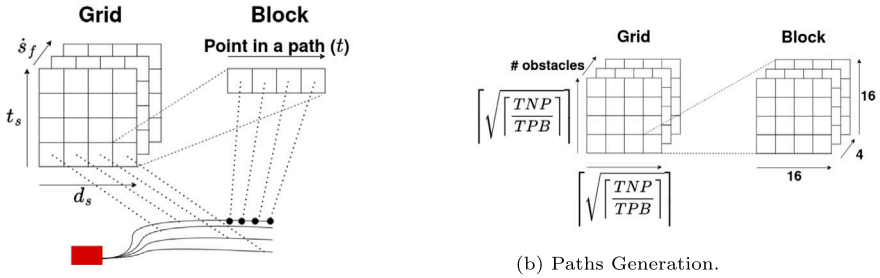
Since all cost components $p.\ddot{s}^2$ and $p.\ddot{d}^2$ must be already computed to calculate the cost C, we exploit the *shared memory* of GPU and CUDA *syncthreads* function. Each thread stores the $p.\ddot{s}^2$ and $p.\ddot{d}^2$ to *shared memory* and, before performing the sum over all points, we call the *syncthread* function. After sync, we are sure that all of the cost parts are calculated so it is possible to make the sum which is performed by only one thread. More precisely, a thread compute all points component d, $\dot{d}$, $\ddot{d}$, $\dddot{d}$, s, $\dot{s}$, $\ddot{s}$, $\dddot{s}$, x and y (in the same way described in algorithm 1). Moreover it computes $p.\ddot{s}^2$ and $p.\ddot{d}^2$. Then the *syncthreads* function is called to ensure that all threads have computed the components. Eventually, a selected thread performs the sum. To perform this sum, the kernel is launched to allocate ($\frac{T_{max}}{T_s} \cdot sizeof(T)$) of *shared memory*, with T being the chosen datatype (*double*, *float*, or *half*).

4.2. Collision check

The *Collision Check* kernel offers two degrees of parallelization: path points and obstacles. Similarly to the previous phase, the mathematical operations in this phase remain unchanged from the original implementation. We improved efficiency by parallelizing the collision checks between path points and obstacles. Additionally, we set a maximum cost for all the colliding paths, hence excluding colliding paths from this selection procedure. These enhancements significantly reduce the computational time required for this phase. Each point in a path must be tested against collisions with every obstacle. Each collision check is independent of the other, so all of these tests can be computed by parallel GPU threads. The kernel is launched spreading all path points on different threads; each block has a fixed dimension of $16 \times 16 \times 4$ threads (resp. on x, y and z axes), for a total of 1024 threads per block (TPB). The grid is composed of an amount of blocks that is adequate to cover the total number of points (TNP) on axes x and y, the axis z is used to map the obstacles (see Fig. 3(b)). The number of blocks for axis x and y is $\left\lceil \sqrt{\left\lceil \frac{TNP}{TPB} \right\rceil} \right\rceil$ while the number of blocks on the z axis is equal to the number of obstacles. The number of blocks of x and y axes are calculated to be sure that all points have an associated thread. Each block can manage TPB points, so the number of blocks must be greater than TNP/TPB. To distribute the needed blocks across both x and y axes, the square root of the blocks' number is computed and rounded up. In this way, it is guaranteed that all points have an associated thread. In the end, the kernel launch configuration is ($\left\lceil \sqrt{\left\lceil \frac{TNP}{16 \cdot 16 \cdot 4} \right\rceil} \right\rceil$, $\left\lceil \sqrt{\left\lceil \frac{TNP}{16 \cdot 16 \cdot 4} \right\rceil} \right\rceil$, #obstacles) for the grid and (16,16,4) for the block where TNP is $\frac{D_{max}-D_{min}}{D_s} \cdot \frac{T_{max}-T_{min}}{T_s} \cdot \frac{V_{max}-V_{min}}{V_s} \cdot \frac{T_{max}}{T_s}$. In this case, the use of *shared memory* is not needed so it is left to zero. In this way, we cover all the possible interactions among obstacles and points. As far as the launch configuration is concerned, we found experimentally that setting it to (16,16,4) as block size is the configuration that minimizes the kernel completion latency.

Each thread performs the collision test as detailed in Eq. (A.8) but only considering one point and one obstacle. If the check fails, then the path of the point is marked as collided and the cost of the path is set to the maximum value to ensure that it will not be chosen as the best path. When all paths are checked, the one with the smaller cost is retrieved using a CUBlas API function call; specifically, by invoking the function *cublasI<T>amin*[3]. This function takes in input an array of

[3] https://docs.nvidia.com/cuda/cublas/index.html#cublasi-lt-t-gt-amin



(a) Paths Generation.

(b) Paths Generation.

Fig. 3. CUDA launch configurations.

values and return the index of the smaller item. The T depends on the type of the array items: in our case, *d* for *double* and *s* for *float* and *half*.

Given that the paths that collide have the maximum cost and the best path choice procedure returns the path with the smallest cost value. If the retrieved best path is marked as collided, it means that there are no feasible paths around the generated trajectories.

5. Experiments

We conducted a comprehensive set of experiments to evaluate our implementation from various perspectives. In this section, we described the hardware platforms tested as well as the experiments aimed at assessing performance in terms of precision and completion times. All these settings are explained in this section and results are shown and commented in Section 6.

5.1. Hardware platforms

We target two hardware platforms for experiments. The first is an NVIDIA Xavier AGX. We performed on this board all experiment instead that requires the F1tenth setup. This embedded board is equipped with a CUDA-capable GPU (512 NVIDIA cores) and an ARM CPU (8 cores). It features 32 GB of DDR SDRAM shared between CPU and GPU. This embedded board is more powerful than the board used in the F1tenth vehicle and it is more representative of the board mounted in real-size vehicles.

The second is the hardware adopted by F1tenth. This hardware is the Nvidia Jetson Xavier NX. This board is equipped with a 6-core ARM CPU, 8 GB of RAM shared between CPU and GPU, and 384-core NVIDIA Volta GPU. This board is a cut-down version of the NVIDIA Jetson Xavier AGX mentioned before.

5.2. Algorithm execution time

We compared our implementation with a publicly available CPU implementation of the *Frenet Planner*[4] as the baseline (CPU); with a parallel CPU implementation that exploits OpenMP (CPUOMP); and with an implementation of [37] that represents the state-of-the-art GPU implementation (GPUSotA).

The execution time of the algorithm is a critical factor in safety-critical systems like autonomous vehicles, where rapid execution is essential for timely reactions to prevent risky situations such as collisions. The primary goal of a planner is to avoid collisions and compute feasible paths. The algorithm's execution time must correlate with the vehicle's speed; higher speeds necessitate faster reaction times. In high-speed scenarios, such as racing, faster algorithm execution allows the vehicle to achieve higher speeds, making it crucial to minimize the planner's execution time. Previous Frenet implementations are able to complete the circuit under test without collisions provided that their speed is limited by the performance of the execution time of the algorithm (Table 1). On the contrary, our solution features an optimized computational time, hence enabling its use at higher speeds and resulting in shorter lap times. We performed the comparison by varying the precision type (*half*, *float*, *double*). These precision types have different data sizes as defined by IEEE standard [45] and this means a different amount of data to exchange between CPU and GPU but also different accuracy in the results.

Table 1
Comparison of *Frenet Frame* based *Path Planners*. Execution time is measured by setting the algorithm to generate 1024 paths and 64 points for each of them.

	[24]	[29]	[37]	Our
Path Generation	CPU	CPU	GPU	GPU
Collision Check	CPU	CPU	no	GPU
Best path choice	CPU	CPU	CPU	GPU
Available code	no	yes	no	yes
Prog. Language	–	Python	C++/CUDA	C++/CUDA
Execution Time (ms)	68.48	4158.89	67.42	7.75

We chose to present a comparison with these implementations because the source code of CPU implementation is available, it implements the algorithm described in the original *Frenet Path Planner* paper [24], and it is written in C++. The parallel OpenMP implementation fully exploits CPU compute power due to parallelism, hence making this a suitable attempt at a fair comparison. The implementation of [37] is the state-of-the-art of GPU implementation. In Table 1, we report a summary of *Frenet Path Planner* implementations available in the literature. The other previous contribution for which the code is made available is [29]. In this case, their code is written in Python, thus performance cannot be fairly compared to our C++/CUDA implementation.

Note that the source code of the state-of-the-art GPU implementation [37] is not available (see Table 1). Moreover, the authors reported performance related to an older embedded system. Hence, the results reported in [37] are not comparable with ours. For these reasons, we have implemented the proposal of [37] to compare the execution time of the entire pipeline (same implementation used in preliminary experiments [6]) on the same board. There are also other differences, such as the fact that they do not exploit GPU for the *Collision check* phase, and the authors split the *Path generation* phase into three kernels: one that computes the paths, one that computes the cost and one that converts the path points from *Frenet coordinates* to *World coordinates*. The authors report the execution times of each kernel separately. In our implementation only one kernel is used to compute paths, cost, and perform the coordinate conversion: by doing so we are able to reduce the kernels' launch overhead, so the reported times include all of these tasks. Moreover, the authors in [37] compute the trajectory selection (based on the cost) on the CPU. This implies the copy of all generated trajectories from GPU to CPU. In our proposal, this selection phase is performed on GPU, hence, only the best selected trajectory is copied to the CPU.

[4] https://github.com/arvindjha114/frenet_planner_agv

We performed two types of measures. One considering the overall processing time (as in preliminary experiments [6]) and one considering the distinct subparts. More specifically, we measured the time of the *Path generation* phase and the time taken by *Collision check* phase separately. Since in our implementation, the kernel that computes the paths (Section 4.1) also performs the cost computation and the conversion of *Frenet coordinates* to *World coordinates*, we compare our *Path Generation* kernel with the sum of the three tasks as described in the baseline implementation. We do not report the CPUOMP and GPUSotA results for these two distinct subparts because the overall results show that they are very close to the baseline CPU implementation, so we focus more on the latter.

In this study, two tests are performed; one varying the number of generated paths while maintaining a fixed path length, and the other varying lengths while maintaining a fixed number of generated paths. For each test, we have performed 100 iterations, and we report the average time and the speedup with respect to the CPU implementation using the same precision type.

The results of these experiments are reported in Section 6.1.

5.3. Ablation study

In the previous experiment, we measured the speedup of each individual phase implemented on the GPU compared to its CPU counterpart. In this experiment, we analyze the impact of porting each phase to the GPU on the overall pipeline, assessing the speedup from input to output when only one phase is offloaded to the GPU. In some instances, porting only a single phase results in increased data transfers between the CPU and GPU, which can affect the overall speedup. Our analysis focuses on the implementation using the *double* precision data type.

The results of these experiments are reported in Section 6.2.

5.4. Algorithm precision

In the first experiment, we measured how different precision types affect the algorithm completion times. Trivially, using fewer bits to represent input and output data leads to a significant reduction in execution times. However, using a less precise data type can compromise the quality of the output trajectory. Therefore, we measured the precision of trajectories generated using *double*, *float*, and *half* data types, similar to our experiments on execution time. Our goal is to identify an optimal trade-off between execution time and the quality of the results. In this experiment, we, therefore, investigate how reducing data type precision affects the quality of the computed trajectories. We aim to measure the trajectory degradation caused by using less precise data types. In addition to execution time, the quality of the trajectory is crucial. A quickly computed but less precise trajectory can still pose risks, as poor trajectory quality may fail to avoid collisions effectively. We compared the trajectories generated using the different data precision formats in both CPU and GPU implementations with respect to trajectories generated by the CPU *double* implementation (henceforth our baseline). We compare paths generated by different precision types but with the same parameters (D_s, D_{max}, D_{min}, V_s, V_{max}, V_{min}, V_{target}, T_s, T_{max}, T_{min}) and the *Frenet* algorithm was set to generate 1024 different paths, each of them with 1024 path points. On top of this, we have simulated a vehicle that follows the selected path but when it reaches a new location it generates again the path to follow. In this way, the generated paths will always feature a different initial state and this is the typical behavior of a vehicle that uses the *Frenet path planner*. We repeat the generation 300 times and we compare each point of each selected path. We have used the Average Trajectory Error (ATE) which is the average displacement error of each point of the path. The displacement error is computed as the Euclidean Distance.

Typically, the vehicle follows the selected path for the first points and then recomputes it based on the new state, *i.e.* its new position; therefore, it is unlikely that the vehicle will reach the ending points that were initially generated. For this reason, it does make sense to measure the average error of the path traveled by the vehicle using the sequence of selected paths computed during the trip. We measured the ATE of the trajectory of the simulated vehicle that moves as described above.

The results of these experiments are reported in Section 6.3.

5.5. Impact of Nvidia UM

The CUDA API exposes two main ways to use data in the GPU context: explicit copies (COPY) and Unified Memory (UM). The first copies all data before kernel execution, and the second copies data only when requested by the computation during the execution of the kernel. The choice between these two methods can affect the software's execution time, so we aim to measure the impact of this decision.

We tested these two approaches in our implementation to find the best way to perform copies. Since the size of data to transfer is important we tested both approaches varying the number of obstacles and consequently, the amount of data to be copied. Moreover, we tested the impact of data prefetching on UM behavior. On the board used in our test, the prefetch behavior is performed by calling the *cudaStreamAttachMemAsync* function[5]. Using this function the driver can optimize the memory coherence operations.

The results of these experiments are reported in Section 6.4.

5.6. Impact of interference

The Frenet Path Planner is only one of the parts of an autonomous vehicle. This means that other processes (such as localization or perception) will execute concurrently with the planner on the same hardware. The execution of more processes on the same board can result in memory interference since more process tries to access it. This behavior can reduce the performance of the planner increasing the execution time. We aim to measure the impact on the planner's execution time when other processes are using the system DRAM memory. This scenario is common in autonomous vehicles, and the resulting slowdown can potentially create risky situations. We tested how our implementation performs in this situation. We are interested in the worst-case scenario so, since the board has 8 cores, we execute our planner with other 7 memory-intensive processes proposed in [46]. We test two variants of interfering processes: one that runs on CPU and generates high-memory traffic using *memset* (CPU interference); the other that uses the CUDA API *cudaMemset* to use the GPU (GPU interference). To reduce the effect of the CPU processes scheduler we pinned each process on a different core.

The results of these experiments are reported in Section 6.5.

5.7. Experiments on the F1tenth setup

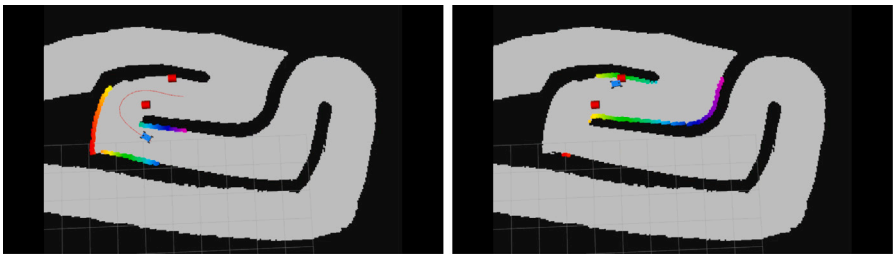
To assess the effectiveness of our approach, we set up a test representative of racing scenarios, namely a racing simulator for 1:10 scale vehicles. We aim to evaluate our implementation on a real vehicle in a racing scenario, with the primary objectives of completing a lap without crashes and achieving maximum speed.

During the F1tenth challenge[6] [47], participants set up an autonomous vehicle in scale, and (among the other trials) must perform collision avoidance of static objects, and overtaking other vehicles in head-to-head sessions. That is where the *Frenet* planner plays a key role, and we validated its effectiveness using the simulator.

In a simulated scenario, we were capable of generating corner cases that stress specific configurations of the path planner. We connected the

[5] https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__HIGHLEVEL.html#group__CUDART__HIGHLEVEL_1g496353d630c29c44a2e33f531a3944d1

[6] https://f1tenth.org



(a) (b)

Fig. 4. F1tenth simulated scenario. Obstacles in red.

Table 2
Parameters used in tests on F1tenth.

D_{min}	D_{max}	D_s	T_{min}	T_{max}	T_s	V_{min}	V_{max}	V_{target}	V_s
−1.0	1.0	0.05	2.0	2.0	0.1	4.5	5.5	5.0	1.0

target embedded board to a workstation running the simulator, hence implementing *Hardware-in-the-Loop (HiL)* configuration, where (negligible) communication delays are traded for timing accurate execution on the real production hardware[7]

All the experiments conducted over the racing scenario were executed with the use of a simulator, which is also the main software environment publicly provided by the F1tenth Autonomous Racing Community. It is called F1tenth gym simulator[8]. This tool provides several fundamental services that can be exploited during any kind of research that involves active testing in a full *Hardware-in-the-Loop* use case. Besides a graphic interface, it automatically manages every communication, perception, map distribution, throttle, and steering so that it is possible to recreate an artificial space optimized and configured to cooperate with the algorithms written by the user. It uses the Robot Operative System 2 (ROS2) [48] to automatically manage the communication between the hardware in which the algorithm executes and the simulator engine. In this way, it is possible to reconstruct an artificial space in the simulator without the algorithm knowing that is a simulated scenario instead of a real setting. Moreover, this simulator features a more advanced simulation engine and physical model that enable a more realistic simulation compared to the test performed in Section 6.3.

The experiments are conducted to evaluate the *Frenet Path Planner* performance on the F1tenth vehicle. For this purpose the parameters are set to typical race values, these parameters are shown in Table 2. It results in the generation of 240 paths, each of them containing 21 points.

We measure the frequency in which the system is able to publish a new trajectory and the maximum speed that the vehicle can sustain before crashing. The first measure tells us how the algorithm is reactive on this board, and the second is useful to estimate the reachable improvement for racing using the novel implementation.

The simulation scenario is set with obstacles placed after a turn as shown in Fig. 4(a). This setup is constructed to test the algorithm in stressful situations since the time needed to detect and avoid the obstacle is very short.

We tested the algorithm in the aforementioned scenario varying the precision type on both CPU and GPU versions. To measure the maximum speed reached by the vehicle, the simulation is repeated increasing the target speed of the algorithm, and consequently of the vehicle, until the vehicle is not able to react in time to avoid the obstacle and crashes (as in Fig. 4(b)). The target speed is initially set to 1 m/s and it is increased by 0.01 m/s at each simulation iteration.

The results of these experiments are reported in Section 6.6.

6. Results

In this section, we present the results of the experiments described earlier.

6.1. Algorithm execution time

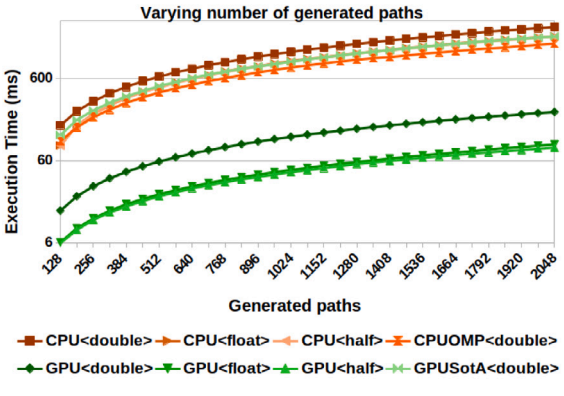
We report the execution time of our implementation and other implementations varying the number of generated paths and the number of points of each path.

The average overall execution times for these two variants are reported in Fig. 5, with Fig. 5(a) displaying the results for the varying number of generated paths and Fig. 5(b) displaying the results for the varying path length. The first consideration, as expected, is that the GPU implementation is significantly faster than the CPU implementation. Moreover, our implementation outperforms the GPUSotA (with a reduction of time of about 8x). This result is also expected since we parallelized the *collision check* phase. The GPUSotA introduces also a huge memory copy overhead and it results slower than the CPUOMP implementation considering the overall execution time. This latter implementation is faster than the baseline CPU implementation but remains slower than our implementation. We report also the speedup of our implementation over the baseline CPU implementation varying the precision types. The results are reported in Figs. 5(c) and 5(d). The precision type impacts the execution time, indeed, the execution time is higher using *double* precision than using *float* or *half*. The execution times of the latter two types are similar but *half* type produces a lower execution time. Indeed, on GPU, using the *float* precision type the times, on average, are reduced by about 74 ms with respect to *double* in the first experiment, and about 36 ms in the second; using the *half* precision type, the reduction is by about 3 ms with respect to *float* in the first experiment, and about 1 ms in the second. The trend of all implementations is linear but the CPU implementation shows a steeper trend due to its serialized processing. In the first experiment (varying the number of generated paths) the speedup with respect to the CPU implementation is constant for *double* (around 11x), increases up to 20x for *float* precision, and up to 22x for *half* precision (Fig. 5(c)). In the second experiment (varying the path length) the speedup is less constant but has the same average values as the first experiment (Fig. 5(d)).

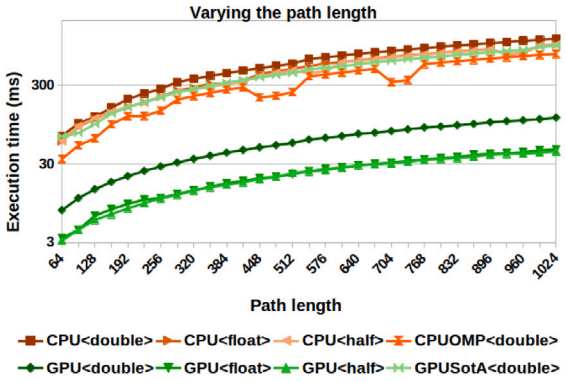
Considering the *Path generation* phase only, the situation is similar. Fig. 6(a) reports the average time of this phase varying the number of generated paths, Fig. 6(b) reports the same times but varying the path length. The speedup of this phase is higher than the overall speedup (Figs. 6(c) and 6(d)): varying the number of generated paths the speedup is around 19x considering *double* precision, 29x considering *float* and 31x considering *half*; varying the path length the speedup for *double* reaches a plateau at a path length of 288 points (around 17x),

[7] This is a typical choice for autonomous systems engineers to test their driving stacks.

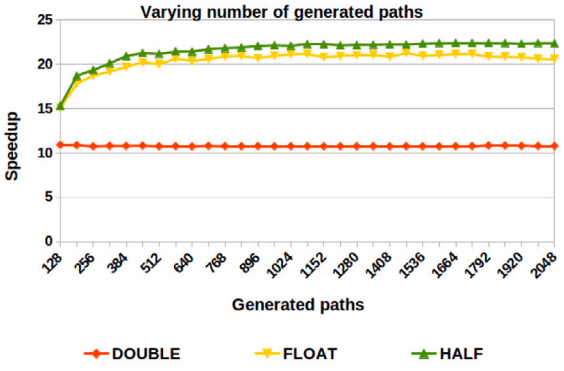
[8] https://github.com/f1tenth/f1tenth_gym_ros



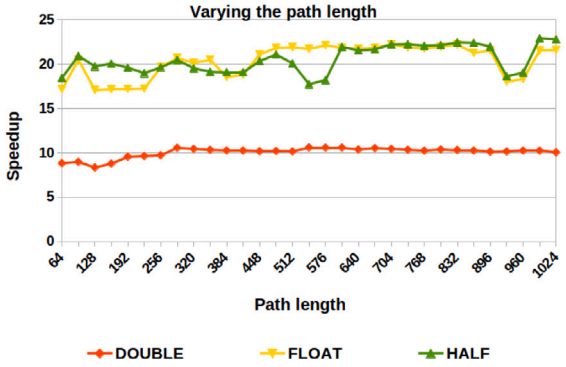
(a) Ex. Time - Varying numbers of generated paths



(b) Ex. Time - Varying the path length



(c) Speedup - Varying numbers of generated paths



(d) Speedup - Varying the path length

Fig. 5. Overall average execution time (ms) and Speedup against CPU.

Table 3
Percentage of phases time over overall time.

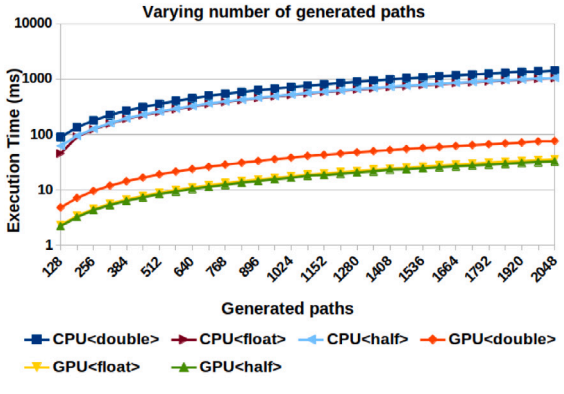
	CPU ⟨double⟩	CPU ⟨float⟩	CPU ⟨half⟩	GPU ⟨double⟩	GPU ⟨float⟩	GPU ⟨half⟩
Path Generation	56.9	53.5	55.0	32.7	38.1	38.3
Collision Check	36.9	44.6	44.5	67.1	61.7	60.7
Other	6.2	1.9	0.5	0.1	0.2	1.0

other precision types have not a constant speedup but always around 25x.

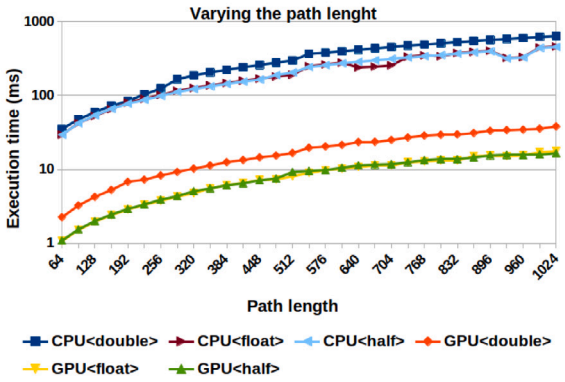
Considering the check collision phase only, the situation is similar. Fig. 7(a) reports the average time of this phase varying the number of generated paths, Fig. 7(b) reports the same times but varying the path length. In this case, the speedup of this phase is lower than the overall speedup (Figs. 7(c) and 7(d)): varying the number of generated paths the speedup is around 6x considering *double* precision, 15x considering *float* and 16x considering *half*; varying the path length the speedup grows until the path length is 224, then it decreases to a plateau around the path length of 544 (around 6x) for *double* type. Other types have fluctuations but the average speedup is about 17x.

The overall execution times and related speedup discussed above also include other phases (*i.e.* the selection of the best path based on the cost). In Table 3 the percentage of the single phases with respect to the overall execution time is reported. In the CPU implementation, most of the execution time is caused by the *Path generation* phase. Instead, in the GPU implementation the highest percentage of execution time is taken by *Collision check* phase. The GPU implementation of *Path generation* phase is constructed exploiting *shared memory* and minimizing the critical operations of a GPU accelerated application, *i.e.* accesses to global memory and branch divergence. Moreover, the merging of *World coordinates* conversion in the same kernel of *Path generation* reduces the overhead of kernel launches. These aspects contribute to the huge execution time reduction of this phase with respect to the CPU implementation. On the other hand, the kernel of *Collision check* phase has an intrinsic branch divergence due to the fact that the instruction flow significantly changes according to whether a collision takes place or not. In summary, the speedup of this phase is lower compared to the speedup of the *Path Generation* phase. This is the cause that the majority of the execution time for the CPU version is spent in the *Path Generation* phase, while for the GPU version, most of the execution time is spent in the *Collision Check* phase.

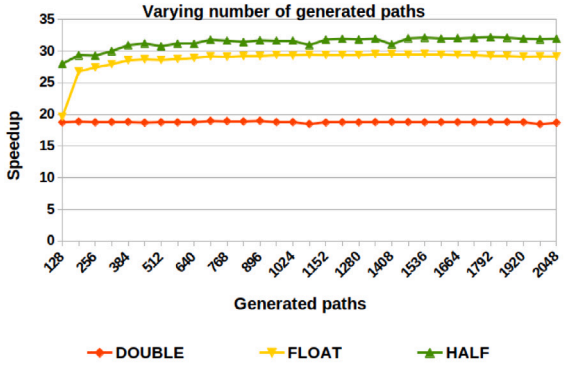
We elected to implement data transfers between the host (CPU) and the device (GPU) using explicit copies to have more control over these operations and be able to execute them in parallel to the kernel execution using *streams*. This implies that there is an additional overhead in the GPU version due to the memory copy of the results to the CPU. This overhead is included in the *Other phases* reported in Table 3. Trivially, in the CPU implementation, the overhead caused by moving data between CPU and GPU is not present, so with the term *Other phases* with regards to the CPU implementation, we refer to the *best path selection* procedure. The table shows that the percentage of *Other phases* is still lower for the GPU implementation. This is due to two factors: first, in our implementation, memory copies are performed concurrently with the kernel computation as we exploited different CUDA streams. In this way, most of the memory copy overhead is mostly hidden by actual computation. Secondly, the *Other phases* also include the computation of the best path (*i.e.* best path choice), that in the GPU case is performed by exploiting the CUDA *cublas* library and results in better performance with respect to the CPU implementation.



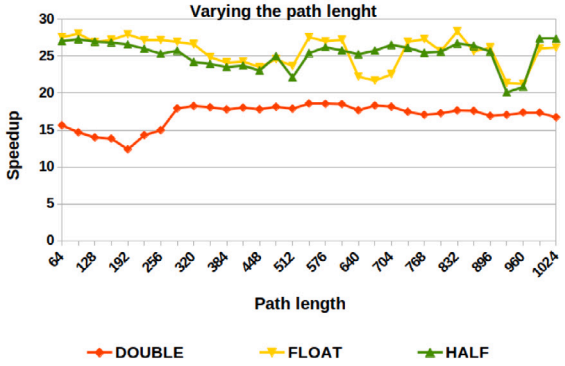
(a) Ex. Time - Varying numbers of generated paths



(b) Ex. Time - Varying the path length



(c) Speedup - Varying numbers of generated paths



(d) Speedup - Varying the path length

Fig. 6. Average execution time of path generation phase (ms) and Speedup against CPU.

Performing tasks on a GPU, such as the best path choice, can reduce the number of memory copies between the CPU and GPU. Indeed, if a task requires input data generated on the GPU by a previous task, performing an intermediate task on the CPU would require two copies that could be avoided by performing the intermediate task on the GPU. Additionally, the final output of a pipeline typically has a smaller size than the intermediate tasks, which means that there would be a smaller amount of data to copy. For example, in the *Frenet Path Planner*, performing the best path choice on the CPU would require copying more trajectory data from the GPU to the CPU, increasing the time required for this operation.

Moreover, the bottleneck task in the pipeline may change when it is implemented on a GPU. This is because different tasks may be more or less parallelizable on a GPU, and GPU implementations may be susceptible to branch divergence and memory access issues. As a result, after the entire pipeline has been ported to the GPU, the bottleneck task may be different than it was in the CPU implementation.

6.2. Ablation study

In this experiment, we assessed the effect of offloading only one phase to the GPU on the overall speedup. Offloading a single phase may involve additional memory copies, potentially resulting in a speedup different from the one reported in the previous section. In Table 4 we report the speedup of the entire pipeline obtained by porting only one phase at a time to the GPU.

Path generation. Offloading this specific phase to the GPU yields the greatest speedup, consistent with our findings from the previous section where this phase demonstrated the highest acceleration. Porting only this phase necessitates an additional copy operation from the GPU to the CPU for the generated trajectory, as the subsequent phase is executed on the CPU.

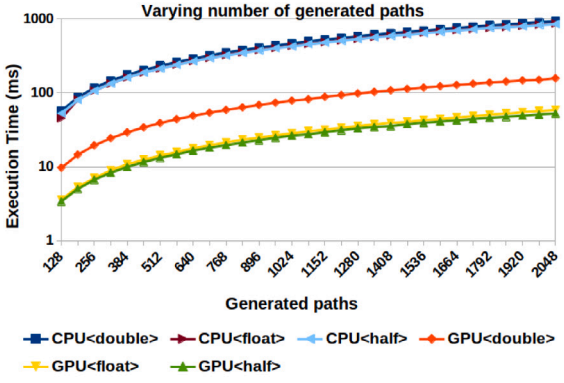
Check collision. Offloading only this phase to the GPU yields minimal speedup. This phase itself achieves a lower speedup compared to the previous phase. Furthermore, porting only this phase involves copying the generated trajectory from the CPU to the GPU, and subsequently copying back the feasible trajectories from the GPU to the CPU, as the first and last phases are executed on the CPU.

Path selection. Offloading only this final phase to the GPU results in minimal speedup. Porting only this phase involves copying the feasible trajectory returned by the previous phase from the CPU to the GPU, and this additional copy operation effectively negates the speedup gained from offloading this phase to the GPU.

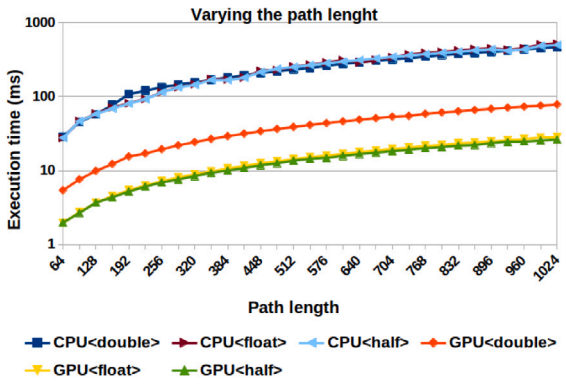
Ultimately, our implementation optimizes the entire pipeline, achieving the highest speedup by offloading all phases to the GPU, which minimizes the need for memory copies.

Considering individual phases, *Path Generation* achieves the highest speedup both within its phase and across the entire pipeline. This phase is predominantly compute-bound compared to the other two phases, which exhibit more memory-bound characteristics (as indicated by the NVidia Nsight Compute Roofline tool[9]), allowing the GPU to effectively leverage parallelism. The *Check Collision* phase, while less compute-bound, consistently operates within the compute-bound region as reported by NVidia Nsight Compute. In contrast, the *Path*

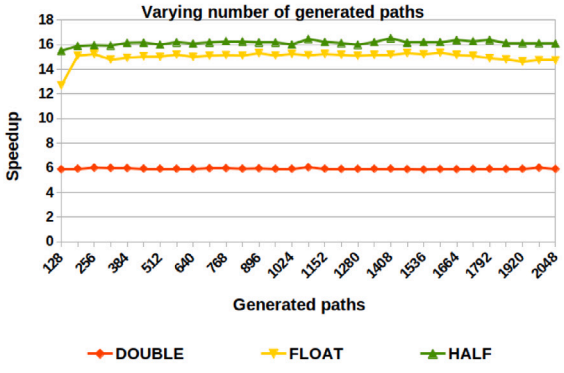
[9] https://docs.nvidia.com/nsight-compute/ProfilingGuide/#roofline-charts



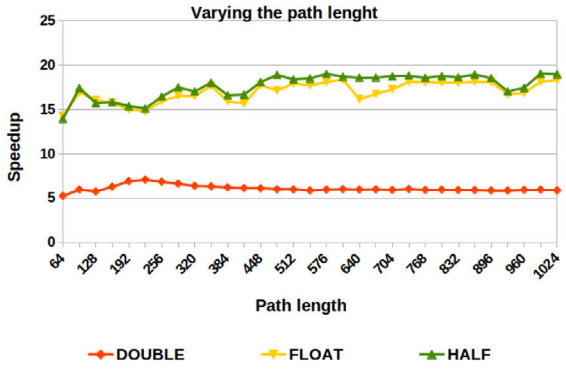
(a) Ex. Time - Varying numbers of generated paths



(b) Ex. Time - Varying the path length



(c) Speedup - Varying numbers of generated paths



(d) Speedup - Varying the path length

Fig. 7. Average execution time of collision check phase (ms) and Speedup against CPU.

Table 4
Overall speedup achieved offloading only one phase on the GPU.

	Speedup
Path Generation	2.5
Check Collision	1.5
Path Selection	1.1

Selection phase, utilizing cuBlas functions, operates within the memory-bound region. The interested reader can see the graphs about roofline model in Appendix B.

6.3. Algorithm precision

In this experiment, we measured the precision error in the output trajectory introduced by using different precision types. Our baseline is the trajectory computed on CPU using the *double* format.

As expected the ATE is 0 m for GPU *double* implementation. On the other hand *half* precision shows a larger error (0.7747 m in GPU and 0,6183 m in CPU) due to the fact that the reduced precision impacts the results. The error in the *float* versions is negligible (0.0005 m in GPU and 0,0027 m in CPU). The small difference between CPU and GPU versions using the same data type is the result of the different hardware architectures.

We also report in Fig. 8 the errors for each point of the path, from 0 to 1024. For each point, we report the mean error measured in each of the 300 paths. We can see that the error within a trajectory increases as the vehicle move further from the path starting point, hence, the precision error tends to maximize towards the last points.

The precision errors are more significant towards those final points, but the trajectory is constantly re-generated, and the path final points are almost never reached. For this purpose, we measured the ATE of the trajectory of the simulated vehicle that moves following the path regenerated when the state changes.

By doing so we obtain the following errors: 0.5993 m for *half* GPU (0,4801 m on CPU), 0.0001 m for *float* GPU (0,0025 on CPU), and always 0 m for *double* GPU.

Qualitatively, the errors reported for *float* and *half* precision do not impact the quality of the generated trajectory due to the fact that the average error is small. Indeed, in Fig. 9 we see the generated trajectory in three scenarios: one in a curve, one in the presence of obstacles, and one in a very sharp turn. Note that the obstacle is dynamic and moving while the algorithm runs. We only report one simulation frame, but we observe that collisions are avoided for the entire simulation. The trajectories are almost the same, but a closer look reveals negligible differences (see Figs. 9(b) and 9(f)). Due to the small magnitude of their difference, it is obvious that even in the *half* precision case, the trajectory is followed and the obstacle is avoided.

To conclude, by decreasing data precision we obtain noticeable performance improvements in terms of execution times (especially using GPU). This comes at the cost of larger ATE values. This outcome is a result of the GPU's intrinsic computational capabilities. Theoretically, given the hardware used in the experiments, two half-precision (FP16) operations can be mapped onto the same floating-point ALU, while a double-precision (FP64) operation requires two floating-point units.

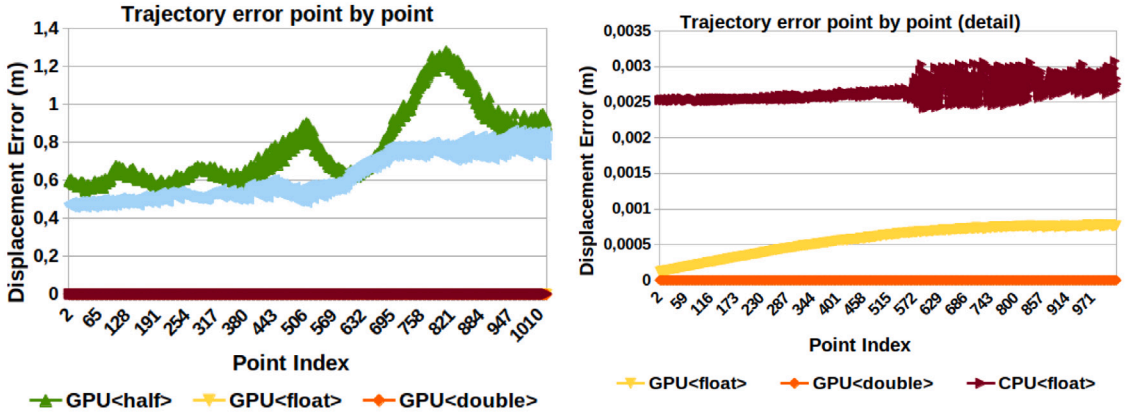


Fig. 8. Trajectory error for each point.

However, this trade-off means that computations using different data types yield variable precision in the results. The choice of precision within data types depends on the context; our experiments, however, clearly show how the *float* version on GPU easily represents the best choice, as it leads to an extremely low ATE compared to the huge execution time reduction (60% of time reduction respect to *double* on GPU). The *half* version shows a higher ATE error towards the final points of generated trajectories, but the performance gain compared to the *float* version is not significant (only 4% of time reduction respect to *float* on GPU).

6.4. Impact of nvidia UM

In this experiment, we measured the impact of using Unified Memory instead of explicit copies.

Results are shown in Figs. 10.

The figures have the overall execution time on the Y-axis and the amount of transferred data on the X-axis.

The execution times of double precision are higher than in the float and half cases but the trend is always the same. The execution time scales linearly with the amount of objects to be transferred. The increase in time depends both on the computational part of the check phase and on the data copy phase.

Both methods (UM and explicit copies) scale in the same way but the UM always shows the worst execution time. This means that the overhead introduced by the UM system is higher than the time needed for the explicit copy. UM system performs data transfer when it is needed if this operation can be overlapped with other computational instructions of the kernel, the overhead can be mitigated by this overlapping. We recall that the majority of data copies regarding the obstacles that are needed in the collision check kernel. Our implementation is optimized to perform this copy during the path generation kernel using two different CUDA streams, so the overhead of explicit copy is mitigated by the concurrent execution of that kernel.

To summarize, in our implementation, the explicit copy is preferable to the UM system since, with the additional control that the programmer has using the explicit copies, we were able to overlap the copies and the kernel execution. Moreover, we found that the prefetching operation does not help the execution time in UM behavior. This is because the UM behavior requires some synchronous operations that are synchronized with other GPU work (*e.g.* other streams) and *cudaStreamAttachMemAsync* is able to reduce the amount of synchronization since it hints the driver that the memory is used only by a stream[10]. In our application, and in particular at the point in which the memory is used, there is only one stream active, so the effect of *cudaStreamAttachMemAsync* is very limited.

6.5. Impact of interference

In this experiment, we measured the slowdown introduced by other processes that run on the same board and use the same memory causing interference.

We report the overall execution time. On the X-axis there is the amount of data copied, and on the Y-axis there is the execution time in milliseconds. We conduct the experiments using all precision types (double, float, and half). Results are shown in Fig. 11.

We can see that interference has an impact on the execution time. Both UM and explicit copy strategies suffer from this behavior so the explicit copy remains preferable also when there are processes that as source of heavy interference.

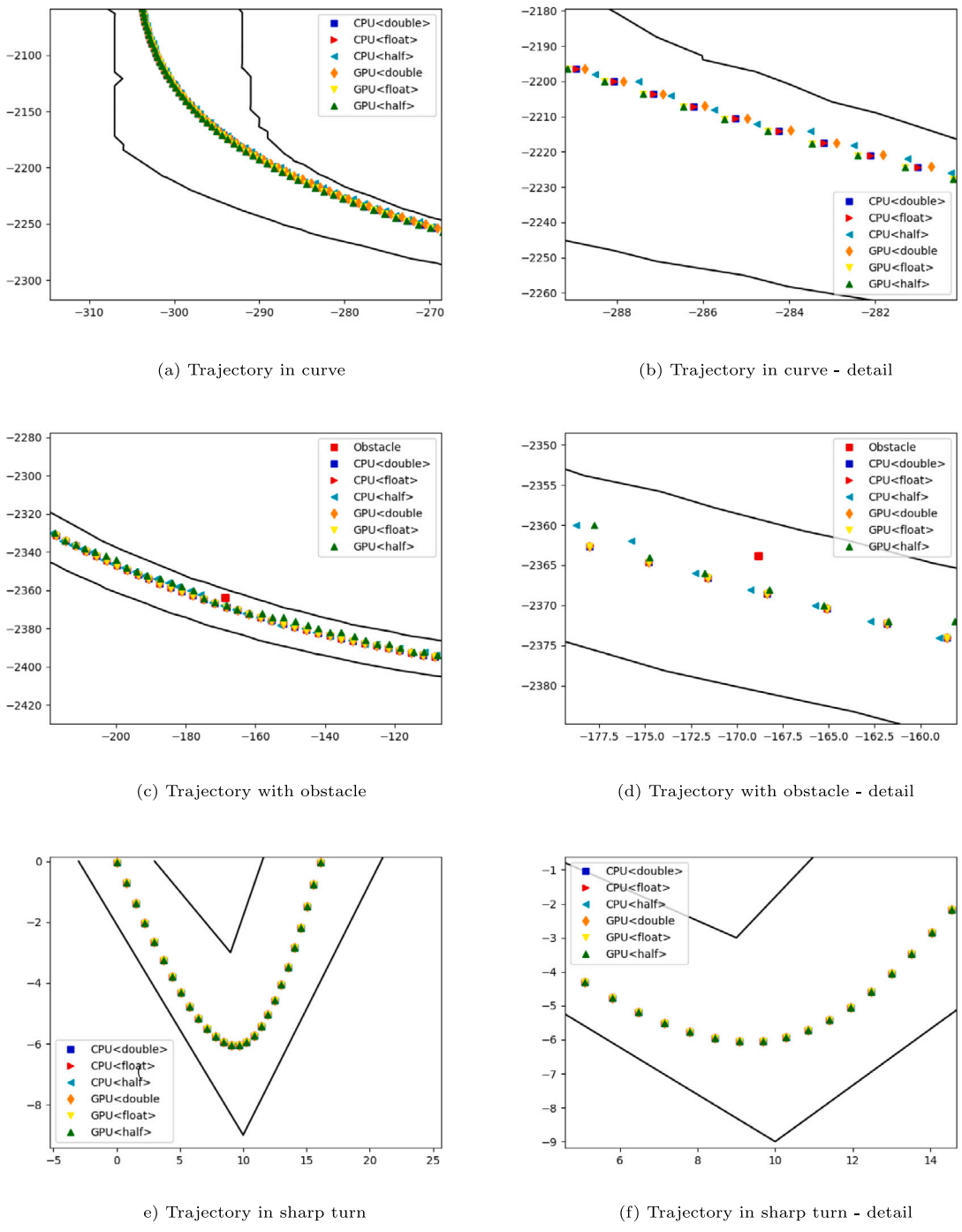
The interference caused by GPU processes has a higher impact on the execution time. Profiling the application with the Nvidia NSight-System tool[11], we see that this is due to the GPU context switch, rather than actual memory interference. A GPU context is spawned from each CPU process that is accessing the GPU, and by default, only one GPU context can reside within the GPU copy or execution engine [39] at a given time. Hence, a context switch is the operation undertaken by the GPU scheduler for selecting which context (*i.e.* GPU application) must be resident in the GPU [49]. In our case, there are different processes that use the GPU, so each of them is scheduled for a fixed timeslice (about 1 ms) in a round-robin fashion. Fig. 12 is a screenshot of the Nvidia NSight-System during the analysis of the concurrent execution of our planner and the other seven interference processes that use GPU. We take the Context analysis line in which green boxes are our planner, grey boxes are other processes. The alternation of contexts is evident and there is no overlapping among contexts. This behavior increases the execution time of our path planner, and for Jetson boards, there is no way to tune the length of such timeslices. Indeed, on Jetson board, unlike the Nvidia DRIVE platform[12], there is not the possibility to set the GPU scheduler policy, the timeslices of each GPU context or processes priority. The only possibility given to the programmer on Jetson boards is to manage the priority of different CUDA streams in the same context. However, there are only two levels of stream priority [39], hence this solution might not be practical for every application scenarios.

The slowdown in time execution caused by interference of the CPU processes is lower. Since system memory in the Jetson board is physically shared between the CPU and the integrated GPU, all processes (CPU and GPU) access the same DRAM banks causing interference. Note that when our path planner runs using the double precision type,

[10] https://docs.nvidia.com/cuda/cuda-for-tegra-appnote/index.html#effective-usage-of-unified-memory-on-tegra

[11] https://developer.nvidia.com/nsight-systems

[12] https://nvidianews.nvidia.com/news/nvidia-introduces-drive-agx-orin-advanced-software-defined-platform-for-autonomous-machines



(a) Trajectory in curve

(b) Trajectory in curve - detail

(c) Trajectory with obstacle

(d) Trajectory with obstacle - detail

e) Trajectory in sharp turn

(f) Trajectory in sharp turn - detail

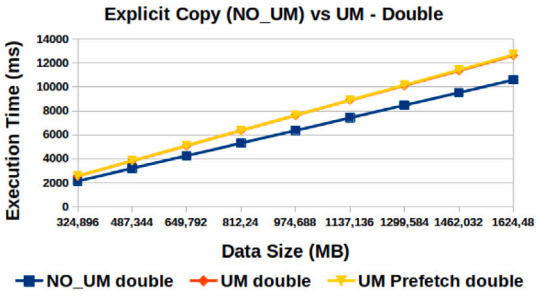
Fig. 9. Generated trajectory: GPU with different data types vs. CPU.

the slowdown is lower. The double computation is more complex and requires more time to be performed, as the Jetson Xavier integrated GPU features half the number of Double Precision (DP) specialized units compared to single precision [50]. This means that the memory is accessed with a lower frequency as operations might be stalled waiting for accessing DP units; since the memory accesses can be executed concurrently with the computation, this results in a lower interference impact.

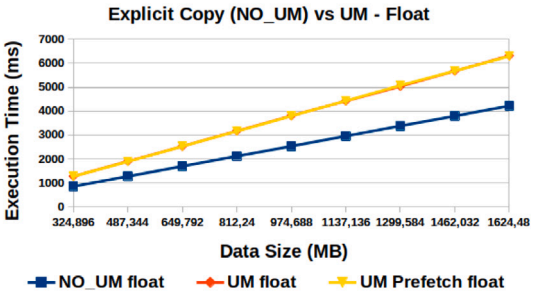
In summary, other processes that run on the same board might affect the execution time of the planner. While CPU co-running processes affect execution times through memory interference, GPU co-running contexts significantly increase the execution time of the observed process due to difficult to manage GPU scheduling mechanisms. In case there is the need to have multiple GPU accelerated processes running on the same autonomous platform, the system engineer must either reorganize all of its GPU implementations within a single process and explicitly schedule streams, or multiple CUDA function calls can be intercepted and scheduled with an ad-hoc software module acting on a separate process [51].

6.6. Experiments on the f1tenth setup

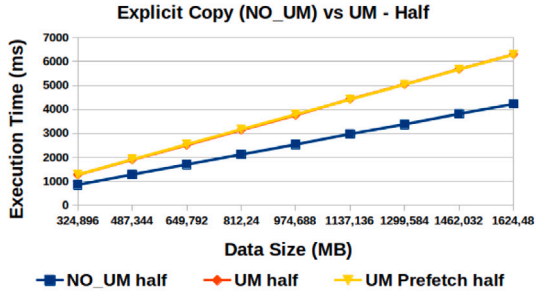
We tested our implementation on the real setup of the F1tenth. We measured the maximum speed the vehicle could reach using our implementation compared to the baseline implementation on the CPU. Moreover, we report the operating frequency reached by the planning node.



(a) Double Precision



(b) Float Precision



(c) Half Precision

Fig. 10. Average overall execution time (ms). Explicit copies vs UM.

In Table 5 we report the results of the experiment.

The reported frequency values follow our expectations and are consistent with the trend shown in Section 6. Our GPU implementation is faster than the baseline CPU implementation and by reducing the precision type, the frequency of trajectory publishing of the algorithm increases. The reported frequencies are related to the entire ROS node execution, so they include the overhead of message reception and transmission. For this reason, the frequencies do not significantly vary if the precision type changes; the execution time differences reported in Section 5 are dominated by the communication overhead, so any further optimization would imply a complete rewrite of the software stack to replace the communication middleware, which is not necessary for our setup, since we already meet our performance requirements, *i.e.*, the sensor frequency.

The most interesting result is the maximum speed the vehicle can achieve before crashing. The unexpected result arises from the fact that, in the GPU version, the highest maximum speed is achieved by the *double* implementation. In the described scenario, a more reactive algorithm would normally perform better by avoiding collisions at higher speeds compared to less reactive algorithms. Consequently, it is expected that the *half* precision version of the algorithm will exhibit the highest maximum speed as seen in the CPU variants. On the other hand, looking at the results of the different precision types on GPU, the maximum achieved speed decreases when the frequency increases. This can be counter-intuitive, but the error introduced by the less precise types must be considered. The error introduced by *float* and *half* precision type reported in Section 6.3 is low and, as reported in the same Section, does not impact the overall trajectory quality. On the other hand, using the F1tenth simulator we are able to see the effective impact of this error in a more realistic scenario. This implies that frequency is an important factor just up to a specific point, beyond which increasing the frequency does not necessarily result in a higher maximum velocity. Rather, the latter factor is primarily influenced by the accuracy of the trajectory. We outperformed our performance constraints (*i.e.*, to 170 Hz), so we can use this extra performance as a "trading token" for maximizing accuracy

Table 5
Result using the F1tenth simulator.

	CPU ⟨double⟩	CPU ⟨float⟩	CPU ⟨half⟩	GPU ⟨double⟩	GPU ⟨float⟩	GPU ⟨half⟩
Frequency (Hz)	98	102	110	170	190	200
max speed (m/s)	2,2	2,25	2,27	2,44	2,36	2,31

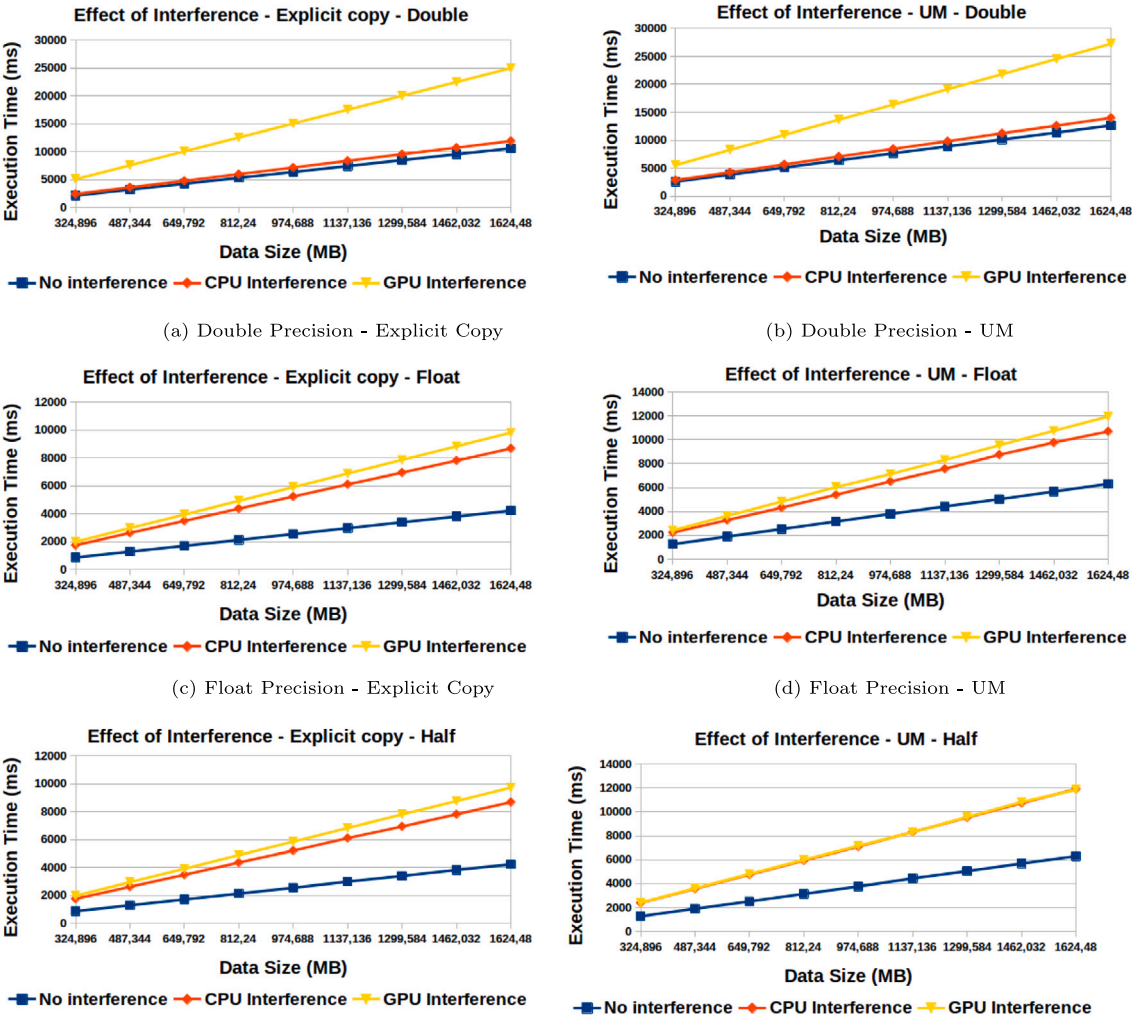
In the end, the reported results show that in the CPU version the vehicle behavior is more affected by the frequency. In the GPU implementation, the error, albeit low, impacts the accuracy of the trajectory of the vehicle. So the increased frequency does not enable the vehicle to avoid sudden obstacles if the generated trajectory is less precise. This leads us to believe that the best speed/accuracy trade-off is achieved by using the *double* version of our GPU implementation, as it leads to a higher computational frequency without sacrificing the trajectory precision.

Moreover, we used our local path planner implementation in a real F1tenth racing. The race was inserted into the International Conference of Robotics and Automation (ICRA) 2023, held in London (UK)[13]. We had two different opportunities to test the planner: during test sessions and the head-to-head competition.

During the head-head competition, two cars raced on the same track. Our planner worked well and we performed several overtaking. In this phase, we won some races and we lost versus the vice-champion for a couple of centiseconds. Moreover, our cars never lost track demonstrating that our implementation achieved very good precision in generated trajectory.

In the test session, all cars are on the same track. This is not a competition phase but, since more cars mean more obstacles, was a

[13] https://www.icra2023.org/ accessed on 22 January 2024



(a) Double Precision - Explicit Copy

(b) Double Precision - UM

(c) Float Precision - Explicit Copy

(d) Float Precision - UM

(e) Half Precision - Explicit Copy

(f) Half Precision - UM

Fig. 11. Average overall execution time (ms) with CPU and GPU interference.

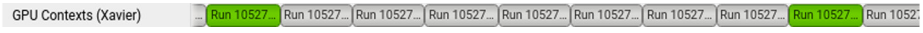


Fig. 12. GPU context switch when more process uses the GPU. Green slots are the path planner process, grey slots are other interference processes.

good scenario to test our local path planner. In this scenario, some cars were stopped while others were running, so there were both dynamic and static obstacles. Moreover, in some cases, cars were very close to each other, making it very difficult to avoid them at high speed. Our car was able to avoid all of these situations making a very difficult slalom demonstrating that our implementation is robust against more obstacles.

Another opportunity to test our planner was in an exhibition race performed at the "We Make Future 2023" in Rimini (Italy)[14]. At this venue, we were able to set up a very large and complex track, with both 15–20 mt straights, and sectors with very sharp turns. On this track, we performed a race with three cars that competed on the same track at the same time. Our car was able to avoid collisions also in this situation and proved it can achieve maximum acceleration in long straights, meaning that the planner can generate the faster trajectory when there are no obstacles.

We report a short video of our planner behavior at this link[15]. Our car is highlighted with the red triangle. The first snippet is an overtake performed in the "We Make Future 2023" exhibition, and the second is an obstacle avoidance demonstration performed during the test laps at the ICRA 2023 race.

Tests in these real scenarios demonstrate that the achieved execution time meets the time constraints of a real racing; indeed our car was able to react promptly when other vehicles were perceived also in tricky scenarios such as blind turns or more cars close to each

[14] https://www.wemakefuture.it/2023/ accessed on 22 January 2024.

[15] https://drive.google.com/file/d/1O8KTBydvntkzGCeDFYesudMBngfTvz_r/view?usp=sharing

other. Some limitations has been found: since the car's sensor cannot perceive obstacles abreast the vehicle, during the overtake the planner can compute a turn to return quickly on the best trajectory. This means that the car will cut the road to the opponent. This is not safe and must be avoided. One solution involves a F1tenth equipment redesign that would introduce more sensors so to be able to cover the blind spots of the car. In this way, the system will be able to see opponents during the overtake. Unfortunately, this is not a possible solution since the F1tenth rules impose fixed equipment for all racing cars. So is not possible to add more sensors and the problem must be tackled using software techniques only. We argue that it is possible to solve the problem by adding opponents' trajectory prediction to the system. In this way, the planner can also consider the next obstacle position when it computes the paths. We plan to implement this in future research.

7. Conclusion

In this work, we proposed a novel and optimized GPU implementation of the *Frenet Path Planner* algorithm. To the best of our knowledge, this is the first implementation that ports the entire algorithm pipeline on GPU. Moreover, we release the source code of our implementation.

We investigated the execution time of our implementation compared to the baseline CPU implementation and we obtain a speedup of up to 22x. Moreover, we investigated the execution time using different data types: *double*, *float*, *half*. Regarding this aspect, we also investigated the impact on trajectory precision when varying these data types and we conducted an ablation study about the contribution of the offloading of each phase to the GPU. Furthermore, we analyzed the impact on the execution time using the Nvidia Unified Memory approach; we then investigated the algorithm performance when other co-running processes were interference on both system DRAM and GPU compute resources.

Finally, we tested our implementation on a simulated racing scenario based on the F1tenth competition but also in two real racing events.

The results confirm that using our implementation with *float* or *half* is convenient since the execution time decreases about 60% with respect to *double* and synthetic experiments suggested that the error over trajectory is not enough to compromise the algorithm efficacy. By testing this in high-speed real world races, however, such as the F1tenth competition, we realized how the sacrificed precision given by the error introduced when moving from *double* to *half* or *float* impact the maximum achieved speed in the presence of obstacles appearing suddenly in the track. In this cases, we showed that using the *double* precision type is the best trade-off.

In future work, we plan to integrate a trajectory prediction functionality able to account for the surrounding vehicles; this constitutes a promising approach in order to manage dynamic obstacles.

CRediT authorship contribution statement

Filippo Muzzini: Writing – review & editing, Writing – original draft, Software, Methodology, Investigation, Conceptualization. **Nicola Capodieci:** Writing – review & editing, Writing – original draft, Supervision. **Federico Ramanzin:** Writing – original draft, Software, Methodology, Investigation, Conceptualization. **Paolo Burgio:** Writing – review & editing, Supervision.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Link to open source code inside article.

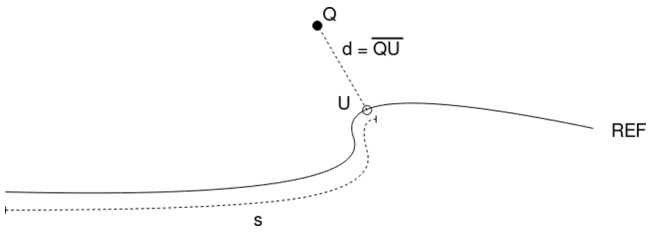


Fig. A.13. Frenet coordinates.

Appendix A. Frenet Path Planner details

A.1. Frenet frame

Considering the position of a point on the curve at time t as $\vec{r}(t)$ and $l(t)$ as the arc length traveled by the point at time t, it is possible to represent t as a function of l using a function $t = f(l)$ so $\vec{r}(t)$ can be rewritten as $\vec{r}(f(l))$; in this way, time is not needed. The *Frenet Coordinates* is formed by $s = \frac{\frac{d\vec{r}}{dl}}{\|\frac{d\vec{r}}{dl}\|}$, that is the tangent unit vector; $d = \frac{\frac{ds}{dl}}{\|\frac{ds}{dl}\|}$, that is the normal unit vector and b, that is the bi-normal unit vector $(s \times d)$.

In the *Frenet Path Planner* s and d assume a different meaning, they are used to represent the *World coordinates* of the point (x and y) in the new *Frenet coordinates*. Considering a point Q in the *World coordinates* (x, y); the s represents the traveled distance on the reference path (REF) and d represents the orthogonal displacement from the Q projection on reference path (see Fig. A.13). The conversion from *World coordinates* to *Frenet coordinates* can be done by projecting the point Q on the curve (calling it U). Because the curve can be seen as a 2D spline function, it is possible to compute the projection and s as the point on the spline that minimizes the distance from Q using the Newton Method. At this point, it is possible to retrieve d as the distance $\overline{QU}$ (see Fig. A.13). In the *Frenet Planner* the reference path can be represented as a 2D spline, that is composed of two splines, one for each dimension (REF_x, REF_y).

A.2. Paths generation

The link that connects the actual state (s_0, d_0) to the final state (s_f, d_f) is a curve and it can be determined by the coefficients of a fifth-degree polynomial for the lateral component (d) and using a fourth-degree polynomial for the longitudinal component (s).

The sampling length of each of these values must be considered as in Eq. (A.1) where $a \in N$.

$$\begin{aligned} \dot{s}_f &\in [V_{min}, V_{max}] \quad | \quad \dot{s}_f = a \cdot V_s \\ d_f &\in [D_{min}, D_{max}] \quad | \quad d_f = a \cdot D_s \\ t_f &\in [T_{min}, T_{max}] \quad | \quad t_f = a \cdot T_s \end{aligned} \tag{A.1}$$

So we can construct the set of possible end states SS_f that include all possible S_f subject to the constraints expressed in (A.1). The path that starts from S_0 and ends in S_f for each $S_f \in SS_f$ is computed solving two different systems (one for d and one for s) as in Eqs. (A.2) and (A.3).

$$\begin{cases} d(t) = d_0 + \dot{d}_0 t + \frac{1}{2}\ddot{d}_0 t^2 + a_3 t^3 + a_4 t^4 + a_5 t^5 \\ \dot{d}(t) = \dot{d}_0 + \ddot{d}_0 t + 3a_3 t^2 + 4a_4 t^3 + 5a_5 t^4 \\ \ddot{d}(t) = \ddot{d}_0 + 6a_3 t + 12a_4 t^2 + 20a_5 t^3 \end{cases} \tag{A.2}$$

$$\begin{cases} \dot{s}(t) = \dot{s}_0 + \ddot{s}_0 t + 3a_3 t^2 + 4a_4 t^3 \\ \ddot{s}(t) = \ddot{s}_0 + 6a_3 t + 12a_4 t^2 \end{cases} \tag{A.3}$$

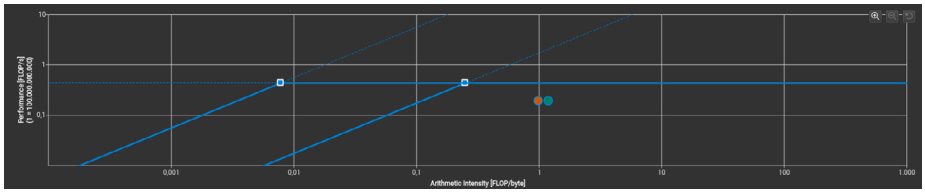


Fig. B.14. Roofline graph of Path Generation kernel.

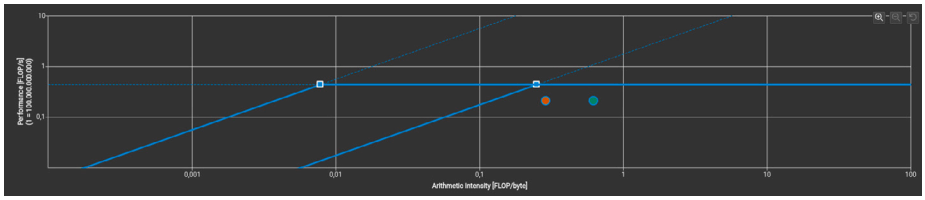


Fig. B.15. Roofline graph of Collision Check kernel.

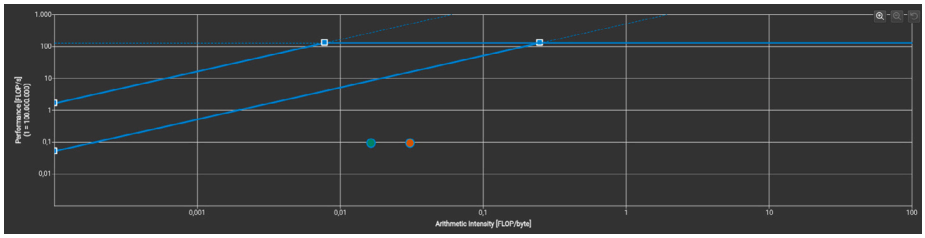


Fig. B.16. Roofline graph of Path Selection kernel (launched by the cuBlas function).

The coefficients a_3, a_4, a_5 in the first system and a_2, a_3 in the second can be found considering the variation of position, velocity, and acceleration from S_0 and S_f. They are obtained by solving the linear systems in Eqs. (A.4) and (A.5).

$$\begin{bmatrix} T^3 & T^4 & T^5 \\ 3T^2 & 4T^3 & 5T^4 \\ 6T & 12T^2 & 20T^3 \end{bmatrix} \times \begin{bmatrix} a_3 \\ a_4 \\ a_5 \end{bmatrix} = \begin{bmatrix} d_f - (d_0 + \dot{d}_0 T + \frac{1}{2}\ddot{d}_0 T^2) \\ \dot{d}_f - (\dot{d}_0 + \ddot{d}_0 T) \\ \ddot{d}_f - \ddot{d}_0 \end{bmatrix} \tag{A.4}$$

$$\begin{bmatrix} 3T^2 & 4T^3 \\ 6T & 12T^2 \end{bmatrix} \times \begin{bmatrix} a_3 \\ a_4 \end{bmatrix} = \begin{bmatrix} \dot{s}_f - (\dot{s}_0 + \ddot{s}_0 T) \\ \ddot{s}_f - \ddot{s}_0 \end{bmatrix} \tag{A.5}$$

Once the functions to retrieve position, velocity, and acceleration for both s and d are constructed, it is possible to discretize the path in f points, so to define a list of path points $P = [p_0, p_1, \dots, p_f]$. Such a discretization is performed with T_s. Eventually, the set of possible paths PS is computed as in Algorithm 1.

Each path $P = [p_0, p_1, \dots, p_f]$ has an associated cost used to determine which of the generated paths is the best. The cost C is computed as in Eq. (A.6).

$$\begin{aligned} J_s &= \sum_{p \in P} p.\ddot{s}^2 \\ J_d &= \sum_{p \in P} p.\ddot{d}^2 \\ d_s &= (V_{target} - p_f.\dot{s})^2 \\ C_d &= k_j \cdot J_d + k_t \cdot t_f + k_d \cdot p_f.d^2 \\ C_s &= k_j \cdot J_s + k_t \cdot t_f + k_d \cdot ds \\ C &= K_{lat} \cdot C_d + K_{lon} \cdot C_s \end{aligned} \tag{A.6}$$

J_s represents the sum of longitudinal jerk and J_d of the lateral jerk. d_s is the squared difference between the end speed and the desired speed V_{target}. C_d is the cost on the lateral side and it is composed of the lateral jerk and the distance of the last point p_f from the reference path. C_s is the cost on the longitudinal side and it is composed of the longitudinal jerk and the speed difference d_s. Moreover, both C_s and C_d considers also the final time of the path t_f. This penalizes long paths in terms of time, indeed, a path with a huge t_f implies that the endpoint will be reached slowly, so it is preferable to a path with smaller t_f. The final cost C is the sum of the lateral and the longitudinal costs. Each cost component has a multiplication factor as a parameter: such a parameter is used to tune the importance of each cost component. In particular, k_j is the multiplication factor for the jerk components, k_t for the time components, and k_d for the displacement components. Moreover, K_{lat} and K_{lon} are the multiplication factors for lateral and longitudinal costs.

The path P is expressed in *Frenet Coordinates*, therefore it must be reconverted back to *World coordinates*. The conversion of a point $p \in P$ is shown in Eq. (A.7).

$$\begin{aligned} I_x &= REF_x(p.s) \\ I_y &= REF_y(p.s) \\ yaw &= atan2(\dot{REF}_y(p.s), \dot{REF}_x(p.s)) \\ x &= I_x - (p.d \cdot sin(yaw)) \\ y &= I_y - (p.d \cdot cos(yaw)) \end{aligned} \tag{A.7}$$

Where REF_x and REF_y are the components of the reference curve as a Spline.

A.3. Collision check

Considering a set of obstacles $OB = \{ob_1, ob_2, \dots, ob_n\}$ and an obstacles radius OR, a path P is considered acceptable if and only if

each path point $p \in P$ has a distance larger than a safe distance SD from each obstacle as in Eq. (A.8).

$$\sqrt{(p.x - ob.x)^2 + (p.y - ob.y)^2} - OR > SD \quad p \in P, ob \in OB \tag{A.8}$$

Eventually, a subset of paths $PS' \subseteq PS$ containing the acceptable paths is constructed. From PS' the best path, based on the cost C computed in Eq. (A.6), is then extracted.

Appendix B. Roofline model

See Figs. B.14–B.16.

Appendix C. Supplementary data

Supplementary material related to this article can be found online at https://doi.org/10.1016/j.sysarc.2024.103239.

References

[1] Sae International, Taxonomy and definitions for terms related to driving automation systems for on-road motor vehicles, SAE Int. 4970 (724) (2018) 1–5.

[2] Ayoub Raji, Alexander Liniger, Andrea Giove, Alessandro Toschi, Nicola Musiu, Daniele Morra, Micaela Verucchi, Danilo Caporale, Marko Bertogna, Motion planning and control for multi vehicle autonomous racing at high speeds, in: 2022 IEEE 25th International Conference on Intelligent Transportation Systems, ITSC, IEEE, 2022, pp. 2775–2782.

[3] Tim Stahl, Alexander Wischnewski, Johannes Betz, Markus Lienkamp, Multilayer graph-based trajectory planning for race vehicles in dynamic scenarios, in: 2019 IEEE Intelligent Transportation Systems Conference, ITSC, IEEE, 2019, pp. 3149–3154.

[4] Micaela Verucchi, Luca Bartoli, Fabio Bagni, Francesco Gatti, Paolo Burgio, Marko Bertogna, Real-time clustering and LiDAR-camera fusion on embedded platforms for self-driving cars, in: 2020 Fourth IEEE International Conference on Robotic Computing, IRC, IEEE, 2020, pp. 398–405.

[5] David González, Joshué Pérez, Vicente Milanés, Fawzi Nashashibi, A review of motion planning techniques for automated vehicles, IEEE Trans. Intell. Transp. Syst. 17 (4) (2015) 1135–1145.

[6] Filippo Muzzini, Nicola Capodieci, Federico Ramanzin, Paolo Burgio, Optimized local path planner implementation for GPU-accelerated embedded systems, IEEE Embedded Syst. Lett. (2023).

[7] Min Seong Kim, Jeon Hyeok Lee, Taek Lim Kim, Tae-Hyoung Park, Frenet frame based local motion planning in racing environment, in: 2023 23rd International Conference on Control, Automation and Systems, ICCAS, IEEE, 2023, pp. 951–957.

[8] Rudolf Reiter, Martin Kirchengast, Daniel Watzenig, Moritz Diehl, Mixed-integer optimization-based planning for autonomous racing with obstacles and rewards, IFAC-PapersOnLine 54 (6) (2021) 99–106.

[9] Bruce Donald, Patrick Xavier, John Canny, John Reif, Kinodynamic motion planning, J. ACM 40 (5) (1993) 1048–1066.

[10] Steven M. LaValle, James J. Kuffner, Randomized Kinodynamic Planning, Int. J. Robot. Res. 20 (5) (2001) 378–400.

[11] Julius Ziegler, Christoph Stiller, Spatiotemporal state lattices for fast trajectory planning in dynamic on-road driving scenarios, in: 2009 IEEE/RSJ International Conference on Intelligent Robots and Systems, IEEE, St. Louis, MO, USA, 2009, pp. 1879–1884.

[12] Ji-wung Choi, Renwick E. Curry, Gabriel Hugh Elkaim, Continuous curvature path generation based on Bézier curves for autonomous vehicles, IAENG Int. J. Appl. Math. 40 (2) (2010).

[13] Julius Ziegler, Philipp Bender, Thao Dang, Christoph Stiller, Trajectory planning for Bertha — A local, continuous method, in: 2014 IEEE Intelligent Vehicles Symposium Proceedings, IEEE, MI, USA, 2014, pp. 450–457.

[14] Volkan Sezer, Metin Gokasan, A novel obstacle avoidance algorithm:"Follow the Gap Method", Robot. Auton. Syst. 60 (9) (2012) 1123–1134.

[15] Mihail Pivtoraiko, Ross A. Knepper, Alonzo Kelly, Differentially constrained mobile robot motion planning in state lattices, J. Field Robotics 26 (3) (2009) 308–333.

[16] Christian Gotte, Martin Keller, Carsten Hass, Karl-Heinz Glander, Alois Seewald, Torsten Bertram, A model predictive combined planning and control approach for guidance of automated vehicles, in: 2015 IEEE International Conference on Vehicular Electronics and Safety, ICVES, IEEE, Yokohama, Japan, 2015, pp. 69–74.

[17] Sterling J. Anderson, Sisir B. Karumanchi, Karl Iagnemma, Constraint-based planning and control for safe, semi-autonomous operation of vehicles, in: 2012 IEEE Intelligent Vehicles Symposium, IEEE, Alcal de Henares, Madrid, Spain, 2012, pp. 383–388.

[18] P. Fiorini, Z. Shiller, Time optimal trajectory planning in dynamic environments, in: Proceedings of IEEE International Conference on Robotics and Automation, vol. 2, IEEE, Minneapolis, MN, USA, 1996, pp. 1553–1558.

[19] Alonzo Kelly, Bryan Nagy, Reactive Nonholonomic trajectory generation via Parametric Optimal control, Int. J. Robot. Res. 22 (7–8) (2003) 583–601.

[20] Matthew McNaughton, Chris Urmson, John M. Dolan, Jin-Woo Lee, Motion planning for autonomous driving with a conformal spatiotemporal lattice, in: 2011 IEEE International Conference on Robotics and Automation, IEEE, Shanghai, China, 2011, pp. 4889–4895.

[21] Wenda Xu, Junqing Wei, John M. Dolan, Huijing Zhao, Hongbin Zha, A real-time motion planner with trajectory optimization for autonomous vehicles, in: 2012 IEEE International Conference on Robotics and Automation, IEEE, St Paul, MN, USA, 2012, pp. 2061–2067.

[22] Thomas Heil, Alexander Lange, Stephanie Cramer, Adaptive and efficient lane change path planning for automated vehicles, in: 2016 IEEE 19th International Conference on Intelligent Transportation Systems, ITSC, IEEE, Rio de Janeiro, Brazil, 2016, pp. 479–484.

[23] Felix von Hundelshausen, Michael Himmelsbach, Falk Hecker, Andre Mueller, Hans-Joachim Wuensche, Driving with tentacles: Integral structures for sensing and motion, J. Field Robotics 25 (9) (2008) 640–673.

[24] Moritz Werling, Julius Ziegler, Sören Kammel, Sebastian Thrun, Optimal trajectory generation for dynamic street scenarios in a frenet frame, in: 2010 IEEE International Conference on Robotics and Automation, IEEE, Anchorage, AK, 2010, pp. 987–993.

[25] Steven M. LaValle, et al., Rapidly-exploring Random Trees: A New Tool for Path Planning, Ames, IA, USA, 1998.

[26] Y. Kuwata, G.A. Fiore, J. Teo, E. Frazzoli, J.P. How, Motion planning for urban driving using RRT, in: 2008 IEEE/RSJ International Conference on Intelligent Robots and Systems, IEEE, Nice, 2008, pp. 1681–1686.

[27] Ulrich Schwesinger, Martin Rufli, Paul Furgale, Roland Siegwart, A sampling-based partial motion planning framework for system-compliant navigation along a reference path, in: 2013 IEEE Intelligent Vehicles Symposium (IV), IEEE, Gold Coast City, Australia, 2013, pp. 391–396.

[28] Luke Fletcher, Seth Teller, Edwin Olson, David Moore, Yoshiaki Kuwata, Jonathan How, John Leonard, Isaac Miller, Mark Campbell, Dan Huttenlocher, et al., The MIT–Cornell collision and why it happened, J. Field Robotics 25 (10) (2008) 775–807.

[29] Majid Moghadam, Gabriel Hugh Elkaim, An Autonomous driving framework for Long-Term Decision-Making and Short-Term trajectory planning on Frenet Space, in: 2021 IEEE 17th International Conference on Automation Science and Engineering, CASE, IEEE, Lyon, France, 2021, pp. 1745–1750.

[30] Michiel J. Van Nieuwstadt, Richard M. Murray, Real-time trajectory generation for differentially flat systems, Int. J. Robust Nonlinear Control: IFAC-Affil. J. 8 (11) (1998) 995–1020.

[31] E. Burns, S. Lemons, W. Ruml, R. Zhou, Best-first heuristic search for multicore machines, J. Artificial Intelligence Res. 39 (2010) 689–743.

[32] Steffen Heinrich, Andre Zoufahl, Raul Rojas, Real-time trajectory optimization under motion uncertainty using a GPU, in: 2015 IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS, IEEE, Hamburg, Germany, 2015, pp. 3572–3577.

[33] Avi Bleiweiss, GPU accelerated pathfinding, in: Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware, 2008, pp. 65–74.

[34] Joseph T. Kider, Mark Henderson, Maxim Likhachev, Alla Safonova, High-dimensional planning on the GPU, in: 2010 IEEE International Conference on Robotics and Automation, IEEE, Anchorage, AK, 2010, pp. 2515–2522.

[35] Ugur Cekmez, Mustafa Ozsiginan, Ozgur Koray Sahingoz, A UAV path planning with parallel ACO algorithm on CUDA platform, in: 2014 International Conference on Unmanned Aircraft Systems, ICUAS, IEEE, Orlando, FL, USA, 2014, pp. 347–354.

[36] Daniele Palossi, Andrea Marongiu, Luca Benini, On the accuracy of near-optimal GPU-based path planning for UAVs, in: Proceedings of the 20th International Workshop on Software and Compilers for Embedded Systems, ACM, Sankt Goar Germany, 2017, pp. 85–88.

[37] Jörg Fickenscher, Sandra Schmidt, Frank Hannig, Mohamed Bouzouraa, Jürgen Teich, Path planning for highly automated driving on embedded GPUs, J. Low Power Electron. Appl. 8 (4) (2018) 35.

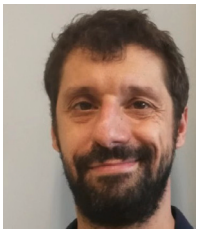
[38] Nicola Capodieci, Roberto Cavicchioli, Andrea Marongiu, A taxonomy of modern GPGPU programming methods: on the benefits of a unified specification, IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 41 (6) (2021) 1649–1662.

[39] Ignacio Sañudo Olmedo, Nicola Capodieci, Jorge Luis Martinez, Andrea Marongiu, Marko Bertogna, Dissecting the CUDA scheduling hierarchy: a performance and predictability perspective, in: 2020 IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS, IEEE, 2020, pp. 213–225.
[40] Yongbin Gu, Wenxuan Wu, Yunfan Li, Lizhong Chen, Uvmbench: A comprehensive benchmark suite for researching unified virtual memory in gpus, 2020, arXiv preprint arXiv:2007.09822.
[41] Jake Choi, Hojun You, Chongam Kim, Heon Young Yeom, Yoonhee Kim, Comparing unified, pinned, and host/device memory allocations for memory-intensive workloads on Tegra SoC, Concurr. Comput.: Pract. Exper. 33 (4) (2021) e6018.
[42] Soroush Bateni, Zhendong Wang, Yuankun Zhu, Yang Hu, Cong Liu, Co-optimizing performance and memory footprint via integrated cpu/gpu memory management, an implementation on autonomous driving platform, in: 2020 IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS, IEEE, 2020, pp. 310–323.
[43] F. Frenet, Sur les courbes a double courbure, J. math. pures appl. (1852) 437–447.
[44] J.-A. Serret, Sur quelques formules relatives à la théorie des courbes à double courbure, J. math. pures appl. (1851) 193–207.
[45] IEEE standard for floating-point arithmetic, 2019, pp. 1–84, http://dx.doi.org/10.1109/IEEESTD.2019.8766229, IEEE Std 754-2019 (Revision of IEEE 754-2008).
[46] Nicola Capodieci, Roberto Cavicchioli, Ignacio Sañudo Olmedo, Marco Solieri, Marko Bertogna, Contending memory in heterogeneous SoCs: Evolution in NVIDIA Tegra embedded platforms, in: 2020 IEEE 26th International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA, IEEE, 2020, pp. 1–10.
[47] Matthew O'Kelly, Varundev Sukhil, Houssam Abbas, Jack Harkins, Chris Kao, Yash Vardhan Pant, Rahul Mangharam, Dipshil Agarwal, Madhur Behl, Paolo Burgio, Marko Bertogna, F1/10: an open-source autonomous cyber-physical platform, 2019, CoRR abs/1901.08567, arXiv:1901.08567, http://arxiv.org/abs/1901.08567.
[48] Anis Koubaa (Ed.), Robot operating system (ROS): the complete reference (volume 6), in: Studies in Computational Intelligence, vol. 962, Springer International Publishing, Cham, 2021.
[49] Nicola Capodieci, Roberto Cavicchioli, Marko Bertogna, Aingara Paramakuru, Deadline-based scheduling for GPU with preemption support, in: 2018 IEEE Real-Time Systems Symposium, RTSS, IEEE, 2018, pp. 119–130.
[50] Hamid Tabani, Fabio Mazzocchetti, Pedro Benedicte, Jaume Abella, Francisco J Cazorla, Performance analysis and optimization opportunities for Nvidia automotive GPUS, J. Parallel Distrib. Comput. 152 (2021) 21–32.
[51] Cheol-Ho Hong, Ivor Spence, Dimitrios S. Nikolopoulos, GPU virtualization and scheduling methods: A comprehensive survey, ACM Comput. Surv. 50 (3) (2017) 1–37.

Filippo Muzzini is a postdoc researcher at the HiPeRT-Lab of the University of Modena and Reggio Emilia, Italy. His scientific interests are about distributed systems and algorithms optimization in parallel and heterogeneous embedded systems with emphasis in the fields of autonomous vehicles and smart cities.

Nicola Capodieci is an associate researcher at the HiPeRT-Lab of the University of Modena and Reggio Emilia, Italy. His main research interests range from distributed systems to languages, architectures and programming models for GPUs with applications in real-time embedded systems.

Federico Ramanzin is a graduate student collaborating with the HiPeRT-Lab of the University of Modena and Reggio Emilia, Italy. His main research interests are about trajectory planning for autonomous vehicle in embedded system.



Paolo Burgio has been with HiPeRT-Lab, University of Modena and Reggio Emilia, Modena, Italy, since 2014. His research interests include next-generation predictable systems based on heterogeneous many-cores and GP-GPUs, with an eye on compilers, and parallel programming models. Burgio has a Ph.D. in Electronics Engineering from the Universitá di Bologna, Italy, and the Université de Bretagne-Súd, France. He is a member of IEEE.