Application and Design of GPU Parallel RRT for Racing Car Simulation

Case Study of Iterative Parallel Sampling RRT applied to The Open Racing Car Simulator

Samuel Simão Canada Gomes

Thesis to obtain the Master of Science Degree in

Computers and Informatics Engineering

Supervisor: Prof. João Miguel de Sousa de Assis Dias
Prof. Carlos António Roque Martinho

Examination Committee

Chairperson: Prof. Daniel Jorge Viegas Gonçalves
Supervisor: Prof. João Miguel de Sousa de Assis Dias
Member of the Committee: Prof. Francisco António Chaves Saraiva de Melo

September, 2017
Contents

1 Introduction 1
   1.1 Work’s Goal .................................................. 1
   1.2 Outline ...................................................... 3

2 Related Work 5
   2.1 Racing Videogame AI ........................................... 5
      2.1.1 TORCS Features Overview ................................... 5
      2.1.2 Existing Bots for TORCS ................................... 5
      2.1.3 Vehicle PID Controller .................................... 7
   2.2 RRT Algorithm .................................................. 8
   2.3 Improving RRT Performance .................................... 8
      2.3.1 RRT* Algorithm ........................................... 9
      2.3.2 GPGPU Constraint Parallelization ......................... 9
      2.3.3 Tree Generation Parallelization .......................... 10
      2.3.4 Distributed RRT ......................................... 12
   2.4 Other RSA Parallelization Methods .......................... 12
   2.5 About GPGPU .................................................. 13
      2.5.1 CUDA Programming Model .................................. 14
         2.5.1.1 Thread Hierarchy .................................... 14
         2.5.1.2 Memory Hierarchy .................................... 14
   2.6 Maintaining a Smooth Framerate ............................. 16
   2.7 Discussion ................................................... 18

3 Iterative Parallel Sampling RRT 21
   3.1 IPS-RRT Description .......................................... 21
   3.2 IPS-RRT Characteristics ....................................... 23
A Appendix: Work Schedule
   A.1 Initial Work Schedule ........................................ 59
   A.2 Work Log ...................................................... 59

B Appendix: Full Test of Section 6.2 ................................ 61

C Appendix: Full Test of Section 6.4 ................................ 63

D Appendix: Code Snippets ........................................ 66
   D.1 RRT Interface (RRT.cuh) ...................................... 66
   D.2 GPGPU IPS-RRT class header (ParRRT.h) .................. 66
   D.3 Sequential RRT class header (SeqRRT.h) ................. 67

Appendix: EPIA Paper ........................................... 69
List of Acronyms

**ALU**  Arithmetic and Logical Unit
**CPU**  Central Processing Unit
**GPU**  Graphical Processing Unit
**GPGPU**  General-Purpose computing on Graphics Processing Units
**TSP**  Traveling Salesman Problem
**TORCS**  The Open Racing Car Simulator
**RSA**  Randomized Search Algorithms
**RRT**  Rapidly-Exploring Random Tree
**MCTS**  Monte-Carlo Tree Search
**PVS**  Principal Variation Splitting
**IPS-RRT**  Iterative Parallel Sampling RRT
**BSD-RRT**  Bulk Synchronous Distributed RRT
**SRT**  Sampling-based Roadmap of Trees
**EST**  Expansive-Space Tree
**PID**  Proportional–Integral–Derivative
List of Figures

1.1 Comparison between the CPU and GPU computing capabilities (in GFlops per seconds) through time. Source (short URL): http://tinyurl.com/z2g29qu (as consulted on Tuesday 12th September, 2017). .................................................. 2

1.2 Screenshot of this work’s bot in action. ................................................................. 3

2.1 Visual representation of the Rapidly-Exploring Random Tree (RRT)* reconnection (top scheme) and rearrangement (bottom scheme) phases. The neighbourhood boundary is represented in blue, the removed/not considered connections in red and the added connections in green. The neighbourhood states have their costs displayed above. In the top scheme, another neighbour is picked rather than the closest one because it has a better (in this case lower) path cost. In the bottom scheme, a connection is changed to improve the cost of an already built path. ................................................................. 10

2.2 Visual representation of each SRT phase. The initial and final states are represented in purple. The trees are represented in blue and outlined in light yellow. The candidate edge computations are represented as dotted brown lines. A tree connection path is represented in green. A discovered path between the initial and final states is painted red. Image inspired by the one at http://www.robotmotionplanning.org/figs/figSRTQueries.jpg (as consulted on Tuesday 12th September, 2017). ............................................. 11

2.3 Scheme of different MCTS parallelizations. Source: [18]. .................................................. 13

2.4 Scheme of the CPU (on the left) and GPU (on the right) architectures. As can be seen, the yellow control area is much bigger on the CPU. Source: [1]. .................................................. 14

2.5 Scheme of a GPU block execution. The blocks ran in parallel are grouped horizontally. A GPU with fewer stream multiprocessors (SMs) will take longer to run all blocks as it has to serialize their execution. The serialized execution order is depicted by the arrows next to the blocks. Source: [1]. .................................................. 15

2.6 Scheme of the GPU thread hierarchy. Two dimensional blocks and their threads are represented. Source: [1]. .................................................. 15

2.7 High level scheme of the GPU memory hierarchy. Each thread has a local memory, each block of threads can access a per block shared memory and the grids can access a global memory. The latency of each type of memory is represented by colour: a low latency memory has a green background and a high latency memory has a red background. Based on a scheme from [1]. .................................................. 17

2.8 Scheme of the GPU memories’ location. Shared memory is built into each multiprocessor. “Device Memory” is a synonym of VRAM. Each stream multiprocessor caches portions of the constant and texture spaces on VRAM. Based on a scheme from [1]. .................................................. 17
3.1 Schematic comparison between an 800 state search tree generated by the sequential RRT version and trees generated by IPS-RRT. The state space is projected to a 2D plane. Black points represent 2D states, black connections represent parent-child relations and the bigger red point is the initial state.

4.1 Diagram of the TORCS bot’s architecture and modules.

4.2 Representation of the bot’s call cycle.

4.3 Detailed scheme of the control module.

4.4 Used method to determine the steering angle. The target search point is represented in blue. The position of the car is displayed in black along with the car direction vector. The angle between the car’s direction and the current state position is displayed in red, as $\theta$.

4.5 Visual identification of the attributes in each state. The darker point represents the car’s current position and the red vector its current velocity; The blue line represents the theoretical trajectory; The brighter point represents the predicted position. The bright red vector represents the predicted velocity; Finally a visual telemetry represents the control parameters (pedals position and steering axis).

4.6 The tree representation used in the IPS-RRT CUDA kernel. Each thread updates part of the array at each iteration. In this example there are three concurrent threads. Therefore, three positions are updated in each iteration. The valid samples can be seen in blue. The ones that were pruned still contain the initialization values.

4.7 Intermediate states and delta representation. $x_{rand}$ is represented in dark blue and $x_{near}$ in light blue. The states’ velocities are also depicted accordingly. Intermediate states are represented in grey. $x_{new}$, represented in orange, is determined by choosing one of the intermediate states. In this case, the third furthest state from $x_{near}$ was chosen. However, the chosen intermediate state is defined initially, just like a delta in RRT.

4.8 Representation of the segments that compose the track. The track segments bounds can be identified in orange. The segment occupied by the car is coloured red. The others are coloured blue.

4.9 Calculation to check if a state lies in one segment. The distances between the sides and the position of the state (represented in blue) are checked to be lower than the distances between the extremes of the track’s segment (represented in brown). If they are, the state is considered to be inside the segment. In this case, the state with position $P_{state}$ lies inside the segment $[P_0, P_1, P_2, P_3]$, because (1) the distances between $P_{state}$ and either $P_{(0,2)}$ or $P_{(1,3)}$ are lower than the distance between $P_1$ and $P_3$, and (2) the distances between $P_{state}$ and either $P_{(0,1)}$ or $P_{(2,3)}$ are lower than the distance between $P_1$ and $P_3$. Although this consists of an approximation, the error is negligible as segments are nearly rectangles (given that The Open Racing Car Simulator (TORCS) tracks generally augment the segment sampling in sharp turns).

4.10 A representation of the line formed by two states on the bezier forward model. The states’ positions are represented in blue and their velocities in red. The parent is the state on the left and the newly generated state the one on the right. The inverse of the newly generated state’s velocity is represented in pink.
5.1 Chart displaying the mean search times for the parallelization test. The bottom part includes the used search parametrizations and the label presents the number of generated states. For example, in an 8 IPS-RRT Iteration search generating 1600 states there are 200 parallel threads (1600/8).

5.2 Chart displaying the mean lap times for the parallelization test.

5.3 Occurrences that can increase lap times.

6.1 Screenshots of the bot's execution in some of the test tracks.

6.2 Layout of the track “CG Speedway number 1”.

6.3 Layout of the track “E-Track 5”.

6.4 Layout of the track “Michigan Speedway”.

6.5 Chart comparing the mean search times of GPU searches ran alongside graphics computations for 800 state searches.

6.6 Chart comparing the mean search times of GPU searches ran alongside graphics computations for 6400 state searches.

6.7 Chart comparing the mean search times of an 800 state search with different number of iterations.

6.8 Chart comparing the mean search times of the GPU implementation of IPS-RRT with different number of states.

6.9 Chart comparing the mean search times of the CPU sequential RRT implementation with different number of states.

6.10 Chart displaying the mean search times for the track “CG Speedway number 1”.

6.11 Chart displaying the mean search times for the track “E-Track 5”.

6.12 Chart displaying the mean search times for the track “Michigan Speedway”.

6.13 Comparison between this solution’s Sequential RRT (on the left) and IPS-RRT (on the right) search tree layouts and path costs for an 800 state search on “CG Speedway number 1”. The blue points constitute the returned path. The accumulated path costs are represented by the number beside such points. It can be seen that the plan generated by IPS-RRT has a bigger path cost (which is better in this case) than the plan generated by the other search.

6.14 Chart displaying the mean lap times for the track “CG Speedway number 1”.

6.15 Chart displaying the mean lap times for the track “E-Track 5”.

6.16 Chart displaying the mean lap times for the track “Michigan Speedway”.

B.1 Chart comparing the mean search times of GPU searches ran alongside graphics computations for 800 state searches.

B.2 Chart comparing the mean search times of GPU searches ran alongside graphics computations for 1600 state searches.
B.3 Chart comparing the mean search times of GPU searches ran alongside graphics computations for 3200 state searches.

B.4 Chart comparing the mean search times of GPU searches ran alongside graphics computations for 6400 state searches.

C.1 Chart displaying the mean lap times for the track “CG Speedway number 1”.

C.2 Chart displaying the minimum lap times for the track “CG Speedway number 1”.

C.3 Chart displaying the mean lap times for the track “E-Track 5”.

C.4 Chart displaying the minimum lap times for the track “E-Track 5”.

C.5 Chart displaying the mean lap times for the track “Michigan Speedway”.

C.6 Chart displaying the minimum lap times for the track “Michigan Speedway”.
Acknowledgements

This work's development benefited from the collection of several fundamental opinions, ideas and critiques. Firstly, I want to thank my coordinators, professors João Dias and Carlos Martinho, who were always keen to help and encourage me in order to produce this work. Both of them were very knowledgeable and secure in their opinions, critiques and feedback. They helped me overcome all kinds of problems from bugs in the case study implementation to spelling errors or even submitting a paper related to this thesis. I think that their help and guidance was key and without it I could not develop such work. Secondly, I would like to thank my friends and colleagues, who helped me to solve some problems associated with the case study implementation and aspects related with the use of CUDA. Thirdly, I would like to thank my parents for all their unconditional patience and support. I could not accomplish anything without them. For all the people who helped and supported me I dedicate this work.
Abstract

Graphical Processing Units (GPUs) have evolved at a large pace, maintaining a processing power orders of magnitude higher than Central Processing Units (CPUs). As a result, the interest of using the General-Purpose computing on Graphics Processing Units (GPGPU) paradigm has grown. Nowadays, effort is being put to study probabilistic search algorithms like the Randomized Search Algorithms (RSA) family, which have good time complexity, and thus can be adapted to massive search spaces. One of those algorithms is Rapidly-Exploring Random Tree (RRT) which reveals good results when applied to high dimensional dynamical search spaces. This work consists in the design, exploration and study of the use of GPGPU-based parallelization techniques in order to improve the application of RRT to racing videogames. To approach such study, a new variant of the RRT algorithm called Iterative Parallel Sampling RRT (IPS-RRT) was developed and a bot for the TORCS open source racing game was built. The results show that, although accesses to the GPU’s memory present high latency, the use of GPGPU-based techniques like the one of this work can still improve not only the planning computational efficiency but also the quality of the returned solutions, as GPU IPS-RRT achieved temporal improvements in big problem sizes (when generating 6400 states) and lap time reductions of around 19%.

Resumo

As Graphical Processing Units (GPUs) têm evoluído rapidamente, mantendo um poder de processamento ordens de magnitude superior ao dos Central Processing Units (CPUs). Como resultado, o interesse em usar o paradigma de programação General-Purpose computing on Graphics Processing Units (GPGPU) tem aumentado. Hoje em dia tem sido revelado empenho no estudo de algoritmos de procura probabilísticos tais como os Randomized Search Algorithms (RSA) que têm uma boa complexidade temporal e consequentemente podem ser utilizados em grandes espaços de estados. Um desses algoritmos é o Rapidly-Exploring Random Tree (RRT), que revela obter bons resultados quando aplicado a espaços de estados dinâmicos com grande dimensionalidade. Este trabalho consiste no desenvolvimento, estudo e exploração de técnicas de parallelização baseadas no GPGPU de modo a melhorar a aplicação do RRT a videogames de corridas. De modo a executar este estudo, foi criada uma variante do RRT chamada Iterative Parallel Sampling RRT (IPS-RRT) e um agente foi construído para o jogo open source TORCS. Os resultados mostram que, embora os acessos às memórias pelo GPU apresentem latências altas, o uso de abordagens melhoradas pelo GPGPU como a apresentada neste trabalho conseguem não só melhorar a eficiência computacional mas também a qualidade das soluções retornadas, dado que o GPU IPS-RRT conseguiu melhorias a nível temporal para problemas de grandes dimensões (gerando 6400 estados) e reduções nos tempos de volta na ordem dos 19%.
Keywords

- General-Purpose computing on Graphics Processing Units
- Randomized Search Algorithms
- Iterative Parallel Sampling RRT (IPS-RRT)
- The Open Racing Car Simulator (TORCS)
- Planning

Palavras Chave

- Programação Geral para Processadores Gráficos
- Algoritmos de Procura com Aleatoriedade
- Iterative Parallel Sampling RRT (IPS-RRT)
- The Open Racing Car Simulator (TORCS)
- Planeamento
Introduction

The fact that Graphical Processing Unit (GPU)’s computing power points to massive parallelization and the fact that this power is still orders of magnitude higher than the Central Processing Unit (CPU)’s (as can be seen in Fig. 1.1), contributed to the development of architectures that can process general information. Such architectures diverge from traditional ones, which exclusively processed graphical elements through matrix calculations. The programming paradigm associated to this technology is commonly called General-Purpose computing on Graphics Processing Units (GPGPU) and was firstly adopted on NVIDIA boards (through the CUDA interface) by November of 2006, according to [1]. As a result of this technological progress, the GPU’s processing capabilities have been used to improve the scalability and execution speedup of many classical search algorithms applying them to a broader variety of challenges connected to AI.

Nevertheless, most classical search approaches lacked the adaptability to problems which generate big continuous state spaces. One example is the space associated with the motion planning problem. The need to search such spaces led to the creation of stochastic search algorithms (algorithms that have a probabilistic approach on search), which only consider representatives of the search space. One example is the Randomized Search Algorithms (RSA) family. Most known RSA include the Rapidly-Exploring Random Tree (RRT) algorithm [20], the R* algorithm [13] (based on A*) or the Monte-Carlo Tree Search (MCTS) [8]. In particular RRT is very interesting, as it can adapt to rapidly changing worlds and high dimensional state spaces, dynamically generating the representation of those spaces. One proposed improvement of RRT is RRT* [2], which can asymptotically converge to better solutions. Other improvements include constraint and tree generation parallelizations ([2] and [17] respectively), or to adapt RRT to distributed contexts [10]. These improvements are detailed in section 2.3.

Many studies on improved versions of known search algorithms, namely RSA, focus on improving their performance by exploring the GPGPU paradigm ([6], [11], [18], [19]). Those studies are generally applied to classical AI problems like the Traveling Salesman Problem (TSP) 1, in board games like Othello 2 or in domains like robot motion planning. However, this approach has not been much explored in real time applications such as videogames.

1.1 Work’s Goal

We believe that videogames can benefit from parallel versions of search algorithms too. For example, racing videogame AI can benefit from the use of parallel versions of RRT, as each state has a big number of variables to be considered like directional speed and steering, positions and physics concepts like velocity or acceleration. On top of that, the generated state space is commonly vast, because of the large number of possible actions that can be taken each moment (considering analogue steering, throttling and breaking).

1http://www.math.uwaterloo.ca/tsp/problem/ (as consulted on Tuesday 12th September, 2017).
2https://boardgamegeek.com/boardgame/2389/othello (as consulted on Tuesday 12th September, 2017).
This work's goal is then to design, explore and study the use of GPGPU-based parallelization techniques to improve the application of RRT to racing videogames. This exploratory study can be divided on three main aspects:

• Computational efficiency of the algorithm: can the use of RRT GPGPU-based parallelization techniques lower the search times?

• Quality of returned solutions: can the use of RRT GPGPU-based parallelization techniques improve the quality of returned solutions?

• Interaction between graphics rendering and the search algorithm: can the same GPU be used to effectively process graphics related procedures along with the search algorithm, or is it just not feasible? Can a smooth framerate be maintained in such cases?

A racing videogame extensively used for academic research continues to be The Open Racing Car Simulator (TORCS)\(^3\), a multi-platform open source game which serves as a base for the construction of racing bots. This game is popular among AI developers, as it presents an API specially conceived for the development of such bots. Besides that, the game is open source which means that programmers have access to its core procedures and therefore can learn what methods do just by analysing their implementations. However, the game was developed such that bots cannot access all of the core procedures. Bots can only view an interface to the track, opponents and game state informations. Examples of such informations include the track drivable boundaries, the current lap number or the last lap time\(^4\). Physics simulation aspects such as the car’s applied forces are not included. The interface is also read only, which means the information it provides cannot be changed. Moreover, the core procedures cannot be used to process non core related data. Nevertheless, such interface seems to be enough for most studies. In spite of the game being around for a considerable amount of time (it celebrates this year its 20th anniversary), it continues to be updated and applied in a wide range of scientific studies ([3], [4], [5]).

\(^3\)http://torcs.sourceforge.net/ (as consulted on Tuesday 12\textsuperscript{th} September, 2017).

\(^4\)Opponents informations such as their positions and velocities can also be checked.
[7] and [8]). Competitions are still held to put new bot ideas to the test\(^5\). Unfortunately, some ideas like the one in this work seldom have the opportunity to compete as these competitions enforce some prohibitory rules about track information acquaintance during the bots executions. Finally, because of its extensive use in competitions and scientific work, TORCS still maintains a big community which is kind to answer developers’ questions. Considering the presented TORCS benefits, this game was used as a case to study (a screenshot of the game running this work’s test cases can be viewed in Fig. 1.2).

It is important to mention that a subproduct of the work here presented (more specifically, the proposal of a new parallel variation of RRT presented and described in chapter 3) led to the acceptance of one article at the EPIA\(^6\) conference. Such article is included in the appendix of this document (it starts at page 69).

### 1.2 Outline

This document is divided into the following chapters: Related Work (starting on page 5) surveys what has been done in terms of AI controllers for TORCS, the studied RRT improvement strategies, what is considered in terms of parallelization for other RSA, some aspects about the GPGPU programming model (more specifically, the CUDA programming model) and what can be considered in order to guarantee a smooth frame refresh rate in graphical demanding applications; Iterative Parallel Sampling RRT (starting on page 21) describes and characterizes the developed RRT variant (as mentioned before, a subproduct of this thesis); TORCS Bot Implementation (starting on page 25) details the solution here presented: its architecture and procedures; Preliminary Tests and Optimizations (starting on page 35) describes some preliminary solutions developed alongside the proposal and throughout the development of the work here presented. This chapter also details some optimizations done to the TORCS bot implementation; Evaluation (starting on page 41) presents and describes the test cases used to analyse this work’s approach, associating the results with the questions underlined in section 1.1; Conclusions (starting on page 55) presents a reflection about what was learned from this work and proposes some possible future work.

---

\(^5\)TORCS competition is an annual event that puts bot implementations side by side. More information about the event can be checked in [http://www.berniw.org/trb/](http://www.berniw.org/trb/) (as consulted on Tuesday 12\(^{th}\) September, 2017).

\(^6\)More information about the event can be checked in [https://web.fe.up.pt/~epia2017/](https://web.fe.up.pt/~epia2017/) (as consulted on Tuesday 12\(^{th}\) September, 2017).
In this chapter, the state of the art is presented. Racing game characteristics are analysed, namely TORCS' related approaches. RRT is detailed along with some of its variants. Some RSA parallelization methods are also described. GPGPU related aspects like the physical architecture of GPU boards or the CUDA programming model are included as they help to better understand such parallelization paradigm. Finally, some improvements to RRT are presented and described along with aspects that have to be considered when dealing with graphical demanding applications.

2.1 Racing Videogame AI

This section introduces some approaches applied to construct TORCS bots, as aspects like the methods used to test the bot or to represent the states were analysed and considered for this work. The Proportional–Integral–Derivative (PID) controller is also described as it was included in this work's development.

2.1.1 TORCS Features Overview

As referred on chapter 1, TORCS presents an API to use in the development of bots. Such API includes attributes that help to describe the car's state and the track. Particularly, control attributes such as the pedals position and steering angle are included to drive the car. The TORCS API is detailed in subsection 4.1.1 because it serves as the interface to the TORCS Back-End.

Moreover, useful content like the simple “bt” bot already implemented in the game (which only follows the centre of the track) can serve as a base for the development of new bots. This work's bot used such implementation as a skeleton.

2.1.2 Existing Bots for TORCS

Although to the author's knowledge no work applied RRT to TORCS, other approaches have been explored. Some appear to simply be based on classical racing line methods. Such approaches include the “bt” driver mentioned in subsection 2.1.1; or the techniques described in [3] and [4], where Bayesian convergence and genetic algorithms were used so that an optimal line could be computed without human intervention (a little bit as an automated racing line method). Nevertheless, more elaborate techniques have been documented, which consider well known planning algorithms. The implementation of the algorithms is not relevant for this work, as such algorithms steps are different from the steps done in RRT. Instead, a more detailed description is given to aspects like the state representations, evaluation

---

1 When AI was input into racing videogames, it was mostly with racing line methods which consisted in pre-drawn lines so cars would follow them.
functions, control architectures or evaluation techniques. That said, such approaches are presented next:

- An Inverse Reinforcement Learning bot [7] was developed with the objective of testing a framework for autonomous road driving. The state representation included features like the lateral displacement with respect to track centre, the absolute speed, the relative speed with respect to traffic limitations and collision distance to an obstacle. A cost evaluation was computed for each of the states. This evaluation was made by a linear weighted combination of the features relative scores. The weights were defined off-line, based on other approaches that were researched by the authors. The plan was built using the Dijkstra algorithm. It consisted in a set of timestamped waypoints. A motion prediction algorithm was implemented. It was described like: (It) "receives tracking data (i.e. position, orientation and velocity) and outputs H grids, representing the posterior probability of the space being occupied at times \{t_1,\cdots,t_K\} in the future". The cost function parameters were estimated by the learning algorithm. They were based on exhibited behaviour (a log file of human driven trajectories). Finally, a path tracking module controlled the vehicle along the waypoints computed in the plan. It was implemented in two submodules:

  - A velocity controller, which inferred the required gear, acceleration and brake commands in order to keep nominal velocity;
  - A pose controller, which sought the points, returning an angle capable of maintaining nominal position and orientation.

This division is very important as it separates the control parameters in two groups: the velocity related attributes and the steering related attributes. This simplifies the controls as it separates attributes that relate to different aspects when driving a car: the gear, acceleration and brake commands define the car’s velocity; The angle returned by the pose controller defines the car’s orientation.

- In [8], MCTS was applied to develop a bot for the TORCS competition (mentioned in chapter 1). The state space was built with a simulation framework. A forward model was included to predict next states and an evaluation function was used to predict the benefit of such states. The used evaluation function is displayed in equation (2.1).

\[
k = \frac{(d - d_0)/(1/2 \cdot a_{\text{max}} \cdot t^2 + v_0t)}{k_{\perp}}
\]

\[
k_{\perp} = k^2 \cdot P
\]

,where \( d \) is the traversed distance along the track line in a given state, \( d_0 \) is the distance along the track in the state at the root of the search tree, \( v_0 \) is the velocity at the root, \( t \) is the time difference, \( k \) is the value in a non-terminal state and \( k_{\perp} \) is the value in a terminal state. \( P \) is a configurable constant between 0 and 1 (the penalty factor).

The state representation was a challenge because, as cited on the article, “(...) the TORCS competition framework only allows access to data that is relative to the cars position on the racing track on any given time step”. Because of that, a forward model used the sensor information obtained by the car which allowed vector-based calculations using the laws of classical physics to simulate future states. This paragraph from the paper helps to better explain the forward model: “The forward model has been implemented by approximating the relationship between the two dimensions of our action space and the car’s position and velocity in Euclidean space, i.e. between pressure on the gas pedal and acceleration, and between rotation of the steering wheel and angular velocity.
These functions have been inferred through experimentation and regression analysis on recorded sets of data.

The excerpts above pointed out the fact that there is not an explicit relation between the velocity of the car and the pedals position. This constraint also revealed to be important while developing this work’s bot. Because of such problem, this work used a PID controller to help minimizing the errors incurred by the forward model. Subsection 2.1.3 presents and describes this type of controllers.

The method used to test and compare this approach was the distance the car drove in a determined amount of time (10000 game ticks: a typical TORCS qualification run). The controller was put side by side with other controllers of 2008 and 2009 TORCS competitions as, in those years, known game tracks were used on the tournament oppositely to dynamically generated tracks. Median score across 10 runs are displayed graphically for three game tracks. In comparison to the 2008 controllers, the approach revealed good results, namely first in the “Street-1” track and hitting top three in “Speedway”. When compared with 2009 controllers, the approach could only manage a place in the last three controllers. The methodology used in this work’s evaluation is very interesting as it uses the typical TORCS qualification in-game time span to compare the bots’ executions. The same time span was used in part of this work’s final tests.

### 2.1.3 Vehicle PID Controller

Proportional–Integral–Derivative (PID) is a control method that can be applied when a direct relation between the control variables and action results of a process cannot be found. It is commonly used in industrial systems. The objective of the technique is to minimize the error found between practical action results and theoretical predictions, getting them closer over time. The value computed in the controller is passed to a system control variable. The control actions are applied to the system through such control variable. The actions taken by a mechanism that implements this type of controller are affected not only by the present error (the proportional component), but also the past errors (the integral component) and future error predictions (the derivative component). All in all, the output of a PID controller is given by a weighted sum of the three components described above. The correspondent formula can be seen in equation (2.2).

\[
    u(t) = K_p e(t) + K_i \int_0^t e(t) \, dt + K_d \frac{de(t)}{dt}
\]

In equation (2.2), \(e(t)\) is the found error (between practical action results and theoretical predictions), \(u(t)\) is the control variable (the output of the controller) and \(t\) is the current time. \(K_p\) represents the weight of the proportional factor, \(K_i\) the weight of the integral factor and \(K_d\) the weight of the derivative factor. These weights are either pre-calculated or dynamically adapted (in that case the controller is called an Adaptive PID Controller) in order to make the system properly react in its environment. Use cases for this method are the stabilization of a rocket [12], or as in this work (and in [14]), vehicle control.

The article [14] describes the PID controller and refers how to generally implement and tweak it so that it can adapt to a racing environment. The article also helps to understand how each weight interacts with the actions taken, the errors that generally occur and how to fix those errors. As cited in that work, the general parameter tweaking process can be viewed like:

1. Set all weights to 0 and increase \(K_p\) until no overshoot and oscillation can be seen (in other words, when the controller achieves a steady state);
2. Increase $K_i$ to fix the steady state error (such error occurs when the controller settles in a state deviated from the desirable);

3. Adjust $K_d$ to reduce overshoot and settling time (settling time represents the amount of time that the controller takes to achieve a steady state).

The author also refers that while in asphalt racing (like in Formula1 or Indy Car) the controller can be tuned for smoothness (with low $K_p$ and small or even negative $K_d$), for other types of racing like in off road racing, $K$ values can be set much larger to speedup the practical effects of the actions.

All in all, the information presented on [14] was useful when building and testing the PID controller for this work’s control module (presented on subsection 4.1.3). The description of the controller served as a base to build it and some advices helped to guide the testing phase.

### 2.2 RRT Algorithm

The algorithm mentioned here was created in the year 1998, documented in [20]. Because of its adaptability to rapidly changing worlds and high-dimensional state spaces, it has been massively applied to robot motion planners. A GPGPU parallel version is referred in [2]. To better understand each of the algorithm steps, it is described bellow along with references to its pseudo-code (represented in Algorithm 1):

- The Algorithm initializes an empty tree with the initial search state (line 2 of the pseudo-code);
- A loop is programmed so that the number of iterations is bounded (lines 3 to 9 of the pseudo-code);
- Inside that loop, a random state is created in the search space (line 4 of the pseudo-code). If the point cannot be created due to a domain constraint, the random state is discarded;
- The tree’s state closest to that random state is picked as the closest neighbour (line 5 of the pseudo-code) and a way of reaching one point to another (an input) is computed (line 6 of the pseudo-code);
- A new state is created by enforcing a variation limit on the random state using the last step input (line 7 of the pseudo-code). Once again, if it is not possible to compute a valid new state, it is discarded;
- The state calculated above is added to the tree and an edge is created between it and the closest neighbour (lines 8 and 9 of the pseudo-code);
- While the iterations limit is not reached, loop again; When the limit is reached the tree is returned (line 10 of the pseudo-code).

### 2.3 Improving RRT Performance

This section describes some techniques used to improve the execution of RRT. First, a variant of RRT called RRT* is presented. Next, other topics like constraint and tree generation parallelizations or distributed RRT are also presented and described.
Algorithm 1 Original RRT

1: procedure GENERATERRT
2: Init tree $T$ with $x_{init}$
3: loop from 1 to $k$:
4:   $x_{rand} \leftarrow \text{randomState}()$
5:   $x_{near} \leftarrow \text{nearestNeighbor}(x_{rand}, T)$
6:   $u \leftarrow \text{selectInput}(x_{rand}, x_{near})$
7:   $x_{new} \leftarrow \text{newState}(x_{near}, u, \delta)$
8:   add vertex $x_{new}$ to $T$
9:   add edge $(x_{near}, x_{new}, u)$ to $T$
10: return $T$

2.3.1 RRT* Algorithm

RRT* (referred and applied in [2] along with RRT) is an optimization of the RRT algorithm to asymptotically converge to a better solution. Its pseudo-code can be seen in Algorithm 2. One difference between RRT* and the original RRT is that in RRT*, a loop checks to see if one of the $x_{rand}$ neighbour states can be part of a path that costs less than the one given by direct choice of the nearest neighbour (loop from lines 9 to 13). Another difference is that the neighbourhood edges are rearranged to reduce path cost (loop from lines 14 to 19). These procedures are represented in Fig. 2.1.

Algorithm 2 RRT*

1: procedure GENERATERRT
2: Init tree $T$ with $x_{init}$
3: loop from 1 to $k$:
4:   $x_{rand} \leftarrow \text{randomState}()$
5:   $x_{nearest} \leftarrow \text{nearestNeighbor}(x_{rand}, T)$
6:   $x_{min} \leftarrow x_{nearest}$
7:   $c_{min} \leftarrow \text{Cost}(x_{nearest}) + \text{Cost}(x_{nearest} \text{ to } x_{rand})$
8:   $X_{near} \leftarrow \text{nearestNeighbors}(x_{rand}, T)$
9: loop for each $x_{curr}$ in $X_{near}$:
10:   $c_{curr} \leftarrow \text{Cost}(x_{curr}) + \text{Cost}(x_{curr} \text{ to } x_{rand})$
11:   if ($c_{curr} < c_{min}$) then
12:     $x_{min} \leftarrow x_{curr}$
13:     $c_{min} \leftarrow c_{curr}$
14: loop for each $x_{curr}$ in $X_{near}$ except $x_{min}$:
15:   $c_{curr} \leftarrow c_{min} + \text{Cost}(x_{rand} \text{ to } x_{curr})$
16:   if ($c_{curr} < \text{Cost}(x_{curr})$) then
17:     remove edge $\{x_{curr}, x_{curr}.parent\}$ from $T$
18:     $x_{curr}.parent \leftarrow x_{rand}$
19: add edge $(x_{rand}, x_{curr})$ to $T$
20: return $T$

A big drawback of RRT* is that it has a big number of dependencies because the additional phases use already calculated information. This makes it much less prone to parallelization than the plain RRT, and so this variation was not considered for this work.

2.3.2 GPGPU Constraint Parallelization

Constraint parallelization, used in [2], helped to improve RRT’s (and RRT**’s) performance by massively parallelizing the process that checks constraints and culls invalid points. As far as the author knows, this seems to be one of the most important works (if not the most important) for the application
Figure 2.1: Visual representation of the RRT* reconnection (top scheme) and rearrangement (bottom scheme) phases. The neighbourhood boundary is represented in blue, the removed/not considered connections in red and the added connections in green. The neighbourhood states have their costs displayed above. In the top scheme, another neighbour is picked rather than the closest one because it has a better (in this case lower) path cost. In the bottom scheme, a connection is changed to improve the cost of an already built path.

of GPGPU to improve RRT’s performance. In that work, the parallelization effort was put to the collision checking procedure which revealed to be the bottleneck (as referred by the authors, 99% of the instructions in the *iterate* method were associated with that procedure). When searching potential object collisions in the work’s motion planning problem, parallel threads checked the collision with each obstacle concurrently. This approach has the drawback of only parallelizing one part of RRT’s execution. If only one obstacle is considered there is no gain because the only thread is going to process the collision of all RRT states. Nevertheless, this technique seems adequate for certain problems where there are a big number of obstacles and therefore a big number of constraints for each state to consider.

2.3.3 Tree Generation Parallelization

A parallelization method called Sampling-based Roadmap of Trees (SRT) [17] creates several parallel independent trees and joins them to produce a global tree which can then be searched to find a solution. SRT can use base planning algorithms such as RRT or Expansive-Space Tree (EST)\(^2\) [9] to create such independent trees. The algorithm works as follows: several samples are taken in the search space. Then, the algorithm uses those samples as the initial node of several trees that can either find a local solution or be joint together to find one. The joining process can use simple paths or more complex techniques like bi-directional RRT or EST. This method pretends to scale algorithms like RRT, providing better performance in vast configurations where the initial and final search states are far apart. The pseudocode associated with this approach is provided in Algorithm 3. A graphical representation of the

\(^2\)EST is a tree generation probabilistic method like RRT. It tries to find a path between two states through expanding them and trying to connect the two subtrees with straight line sub-paths.
Algorithm 3 SRT

1: \textbf{procedure} \textsc{Generate SRT Roadmap} \\
2: \textbf{Input:} $K$ number of trees, $N_c$ number of nearest neighbors, $N_r$ number of random neighbors \\
3: \textbf{Output:} A roadmap $G = (V, E)$ \\
4: \hspace{1cm} $V$ ← empty set; $E$ ← empty set. \\
5: \hspace{1cm} \textbf{loop while} $\text{cardinality}(V) < K$ \textbf{do} \\
6: \hspace{1.2cm} $s$ ← sample collision-free configuration at random \\
7: \hspace{1.2cm} $T$ ← build tree rooted at $s$ using a tree generation algorithm like RRT or EST \\
8: \hspace{1.2cm} add the tree $T$ to the vertex set $V$. \\
9: \hspace{1cm} $E_c$ ← empty set. \\
10: \hspace{1cm} \textbf{loop for all} $T$ in $V$ \textbf{do} \\
11: \hspace{1.2cm} $S_{close}$ ← the $N_c$ closest trees in $V$ to $T$. \\
12: \hspace{1.2cm} $S_{random}$ ← a set of $N_r$ random trees in $V$. \\
13: \hspace{1.2cm} add to $E_c$ all the edges $(T, T')$ for $T'$ in $S_{close}$ or $T'$ in $S_{random}$. \\
14: \hspace{1cm} \textbf{loop for all} candidate edges $(T', T'')$ in $E_c$ \textbf{do} \\
15: \hspace{1.2cm} \textbf{if} $\text{!connected}(G, T', T'')$ and $\text{canConnect}(T', T'')$ \textbf{then} \\
16: \hspace{1.2cm} add the edge $(T', T'')$ to $E$. \\

Algorithm 3 receives a number of trees to be computed ($K$) along with a number of nearest and random neighbours to consider ($N_c$ and $N_r$). $N_c$ and $N_r$ are used when merging each tree in the roadmap $G$. The algorithm works as follows: First, in the lines from 4 to 8, $K$ valid random states are picked and trees are built using such states as initial states. Such trees are stored in the set $V$. Then, in the lines from 9 to 13, for each tree $T$ in $V$, two subsets are computed, $S_{close}$ and $S_{random}$. While $S_{close}$ contains the closest trees in respect to $T$, $S_{random}$ contains a set of random trees in $V$. A distance metric defines closeness between trees. Random neighbours are used to offset any problems with the distance metric\textsuperscript{3}. All edges from $T$ to a tree either in $S_{close}$ or $S_{random}$ are picked as candidate edges.

\textsuperscript{3}As referred in http://www.robotmotionplanning.org/ResearchSRT.html (consulted on Tuesday 12th September, 2017).
(lines from 9 to 13). Those candidate edges are then traversed. Every time one edge is valid (the two edge states can be connected and such relation is not already found), it is added to $G$ (lines from 14 to 16). In the end, $G$ contains not only the generated trees, but also the edges that connect such trees with each other.

### 2.3.4 Distributed RRT

Other approaches for parallelizing RRT are presented in [10]. They are distributed approaches which serve to parallelize the building of a single RRT tree in a multi-process context. One important variant is Bulk Synchronous Distributed RRT (BSD-RRT). BSD-RRT's pseudocode is presented in Algorithm 4.

**Algorithm 4 BSD-RRT**

```plaintext
1: procedure BSD-RRT
2: Input: An environment $env$, a root $q_{root}$, the number of nodes $N$, a stepsize $\delta q$, the number of processes $p$, the number of local expansion steps $m$
3: Output: A tree $T$ containing $N$ nodes rooted at $q_{root}$
4: $T$.AddNode($q_{root}$)
5: loop for all proc $p \in P$
6: $i \leftarrow 0$
7: loop while $i < N/p$
8: localContainer $N_m$
9: loop for $j = 1...m$
10: $q_{rand} \leftarrow GetRandomNode(env)$
11: $q_{near} \leftarrow FindNeighbor(T, q_{rand})$
12: $q_{new} \leftarrow Extend(q_{near}, q_{rand}, \delta q)$
13: if $!TooSimilar(q_{near}, q_{new}) \land IsValid(q_{new})$ then
14: $N_m$.Insert($q_{near}, q_{new}$)
15: loop for all node pair $n \in N_m$
16: $T$.AddNode($n.q_{new}$)
17: $T$.AddEdge($n.q_{near}, n.q_{new}$)
18: $i \leftarrow i + 1$
19: return $T$
```

In this algorithm, processes create several RRT samples and push them to a global tree, which is then distributed to the other processes. In other words, when each processor $p$ stops generating a fixed number of $m$ samples, it updates a tree and propagates the updated tree to each of the other processors through message broadcasting. Approaches such as the ones in [10] reveal to be mostly applied in robot motion planning contexts, where a distributed system makes sense and reveals to be very effective. The same processes cannot be applied in parallel contexts, where the communications between threads are much more direct and straightforward. In those contexts, the idea of parallelizing the creation of a tree can be applied by iteratively synchronizing threads that create the states. In chapter 3, the parallel search algorithm developed in this work, Iterative Parallel Sampling RRT (IPS-RRT), is presented, described and compared to BSD-RRT.

### 2.4 Other RSA Parallelization Methods

As already mentioned, parallelization (namely through GPGPU) is being applied to speedup the execution of RSA in order to apply them to bigger search spaces. The most important parallel versions of RSA (other than RRT) are presented next:
• MCTS ([5],[15],[18] and [21]), which comprehends four phases: Selection, Expansion, Simulation and Backpropagation. The Simulation is the most parallelizable phase as playouts (starting at a specific node, playing the game randomly until a limit is reached) have little dependency with each other. The algorithm typically supports three ways of parallelization: leaf, root or block. A brief description of each method is presented next and a schematic representation can be viewed in Fig. 2.3:

- Leaf parallelization: Play parallel playouts in each tree leaf;
- Root parallelization: Build and search in different parallel trees (augments the search space in relation to the above, compromising each single tree search performance);
- Block parallelization: Mix between the two methods above (in order to augment the search space and at the same time get an improved performance in each tree).

Another method was proposed in [15], the pipeline method, in which each step of the MCTS algorithm is given a block of time and a schedule is built with blocks representing different nodes, forming a pipeline.

• R* (which is a randomized algorithm based on A*) constructs a high level graph by picking random states, executing small A* searches to those states and then a search on the high level graph.

The proposed algorithm is described and GPU parallelized on [11]. It concurrently executes short range searches until the goal node is expanded.

2.5 About GPGPU

Circuit wise, a GPU is a very different chip from a CPU, because the purpose is completely different. While CPU’s Arithmetic and Logical Units (ALUs) are built to deal with complex operations and control flags (for ex. when reading instructions in user and system modes), a GPU deals with a massive number of simple operations executed at the same time. In other words, at a given moment, while a CPU can process some complex operations, a GPU processes a massive number of simple operations. It is for that reason that the control unit on a CPU is so much bigger compared to a GPU’s unit. To clarify such difference, a diagram of both architectures is shown in Fig. 2.4.

4For a more detailed explanation of this algorithm, the viewer can visit http://www.cameronius.com/research/mcts/about/index.html (as consulted on Tuesday 12th September, 2017).
5The reader can check http://techterms.com/definition/alu (as consulted on Tuesday 12th September, 2017) for a definition of ALU.
As a result of the presented GPU's characteristics, GPGPU implies a specific programming model. That model is mapped by different toolkits such as CUDA\textsuperscript{6} or Open CL\textsuperscript{7}. This work uses CUDA because it provides more features for Nvidia cards such as the one used in the evaluation phase (more details about the hardware used to test this work are provided in the beginning of chapter 5). The CUDA programming model is detailed in the next subsection because of the importance it had for this thesis.

### 2.5.1 CUDA Programming Model

#### 2.5.1.1 Thread Hierarchy

A GPU is divided into several multiprocessors. Each multiprocessor can launch several threads. Unlike what happens with CPU threads, GPU threads can be coupled into two or three dimensional blocks and blocks can be coupled into grids. Blocks are simply abstractions which represent groups of threads. As said in [1], "(...) the number of thread blocks in a grid is usually dictated by the size of the data being processed or the number of processors in the system, which it can greatly exceed". Also, threads in different blocks are ran in different stream multiprocessors. As cited on [1], "(...) all threads of a block are expected to reside on the same processor core and must share the limited memory resources of that core. On current GPUs, a thread block may contain up to 1024 threads". Consequently, it is assumed that the blocks execution can be serialized if the data processed by the blocks exceeds the data that can be processed in the card's multiprocessors. A GPU block execution scheme can be consulted in Fig. 2.5.

Threads in the same block can share data through shared memory and synchronize their accesses through synchronization methods. A thread hierarchy scheme can be found in Fig. 2.6.

#### 2.5.1.2 Memory Hierarchy

The programs' data is normally stored on the heap and stack spaces present in the computer's main memory (RAM). However, when using GPGPU, the data processed by the GPU needs to be copied to the graphics card memory, as GPUs use a special hierarchy and cannot directly access main memory. Such memory hierarchy consists in several modules which can be exploited. Each thread has its local memory, each thread block has its shared memory and all blocks can access a global memory. Additionally, constant and texture memory spaces exist for special uses.

\textsuperscript{6}http://www.nvidia.com/object/cuda_home_new.html (as consulted on Tuesday 12\textsuperscript{th} September, 2017).

\textsuperscript{7}https://www.khronos.org/opencl/ (as consulted on Tuesday 12\textsuperscript{th} September, 2017).
Figure 2.5: Scheme of a GPU block execution. The blocks ran in parallel are grouped horizontally. A GPU with fewer stream multiprocessors (SMs) will take longer to run all blocks as it has to serialize their execution. The serialized execution order is depicted by the arrows next to the blocks. Source: [1].

Figure 2.6: Scheme of the GPU thread hierarchy. Two dimensional blocks and their threads are represented. Source: [1].
Global memory is the one which has the biggest latency (as referred in [1] about 300 clock cycles). The data transfer buses have a limited bandwidth (usually up to 150 GB/s, depending on hardware). VRAM is located outside the multiprocessors, which explains such characteristics. The global memory size depends on the amount of available VRAM.

Per block shared memory is associated with the GPU’s cache. It is built directly into each stream multiprocessor. Shared memory has low latency (as referred in [1] about 11 clock cycles) and high bandwidth (usually up to 1 TB/s, depending on hardware). It is used when the data can be divided into chunks. Data has to be transferred from global to shared memory in order to be used between different kernel calls, as this kind of memory is not persistent. The biggest concern when using this type of memory is that it is limited in terms of capacity (in modern devices, from 48 to 49 kilobytes) and so it cannot be used when data chunks are big.

As presented before in this subsection, there are two special kinds of memory, the constant and texture memories. Constant memory is used to store constant symbols. It resides in VRAM but it has one particularity, it is cached. Because of that, the latency is reduced everytime there is a cache hit. This memory can be applied when data is concurrently accessed by a big number of threads. However, like shared memory, it is also limited in terms of capacity (in modern devices, to about 64 kilobytes). Another special memory is the texture memory. Like constant memory it consists in a cached VRAM space. It can be used when the problem involves lots of spatial calculations, because such calculations are done directly by the texture manipulation hardware. Moreover, it can be used when there is thread spacial locality (close threads access close regions in memory) and memory accesses are not predictable. In CUDA, the API for using such memory differs majorly from the API used in the other kinds of memory and therefore its adaptability to certain problems is harder.

To help understand the different kinds of GPU memory, a high level memory hierarchy scheme can be seen at Fig. 2.7 and the location of GPU memories is depicted on the scheme of Fig. 2.8.

### 2.6 Maintaining a Smooth Framerate

Maintaining a good frame refresh rate on graphical demanding applications (measured in frames per second) is very hard because for each time a GPGPU kernel is called there is an interruption on the application’s core execution. Sometimes the applications’ simulations are not influenced by the frame refresh rate, being only dependent on the real execution time. In such cases, interruptions to the application core processing produce jumps. In other cases, the simulation is fully dependent on the frames computed per second. In those cases, interruptions to the application core processing produce execution slowdowns. Therefore interruptions on graphical demanding applications’ internal simulations should be as fast as possible to avoid generating either jumps or slowdowns.

This work focuses on the TORCS videogame which is of the first type presented. The game simulation is frame independent, which means that while driving ahead, interruptions to the game core processing produce jumps in the car’s position instead of smooth transitions. Such jumps have to be minimized in order to maintain graphical performance.

---

8. This fact is very common among nowadays videogames and it is even considered a good practice in most computer games. The objective of the technique is to avoid the hardware to dictate the processing speed, as there is a big variety of hardware that can run the game.

9. This generally happens in fighting games which need to be frame dependent because events like button pressing have to be checked frame by frame.
Figure 2.7: High level scheme of the GPU memory hierarchy. Each thread has a local memory, each block of threads can access a per block shared memory and the grids can access a global memory. The latency of each type of memory is represented by colour: a low latency memory has a green background and a high latency memory has a red background. Based on a scheme from [1].

Figure 2.8: Scheme of the GPU memories’ location. Shared memory is built into each multiprocessor. “Device Memory” is a synonym of VRAM. Each stream multiprocessor caches portions of the constant and texture spaces on VRAM. Based on a scheme from [1].
A good benchmark for the time that can be spent on simulation interruptions is to apply a frame refresh rate considered to be the smooth. This aspect is very subjective and has many interpretations, but there seems to be a consensus that below 24 frames per second (≈ 0.0417 of a second per frame), low rate artefacts start to be notorious on modern devices¹⁰. This work studies the implementation of search algorithms which are executed on a single frame. To ensure a proper plan in such time spans can be quite tricky as a sufficient number of search states have to be processed each step. It can therefore be concluded that for increasing AI’s performance on graphical demanding applications, the information to be generated by each search has to be balanced: It cannot be too low as it does not lead to a good AI performance, and it cannot be too high as it does not lead to a good graphical performance (consistent frame refresh rate). The program data has got to be tweaked based on the hardware specifications, as good hardware can process more information per time span. This work analyses how to exploit the search in order to produce good solutions in a practical time span.

Augmenting the problem size (i.e. the amount of computation in each sequential search) can help to improve the AI planning quality. Using massive concurrent processing power through GPGPU seems to be interesting as the search times do not escalate as quickly. However, it has to be remembered that the executions of graphical and general information have to be balanced in order to produce effective results, and that achieving such performance increases in low time spans is hard, because not only thread creation and synchronization overhead can majorly decrease the efficiency, but also the GPU’s clock rate is lower than the CPU’s. Moreover, as observed while developing this work, high GPU global memory latency can significantly reduce the efficiency of the parallel program when considering such time spans.

2.7 Discussion

The inspiration for the approach presented in this work was the description of the RRT algorithm explored in [2] and [20]. After analysing the parallelization of other searches, RRT was chosen as the case to study, due to its simplicity and the fact that its parallelization on time constraint applications still lacked some exploration, in opposition to classical RRT applications like in robotic motion planning or distributed multi-process contexts.

As racing games constitute problems that imply dynamic attributes (like position and velocity) and time dependent constraints like the equations of motion, the use of the RRT algorithm family seemed to be adequate. TORCS was chosen as the case to study because of the importance it represents to the development and improvement of racing AI techniques through the creation of bots.

The GPGPU tool kit used in this work is CUDA, because as mentioned on section 2.5, it has better performance and also provides more features for Nvidia cards.

SRT was not used to improve RRT, because a big tree expansion generating a road map was not necessary. Instead, as the search was recalculated whenever needed (on the fly), RRT alone proved to sufficiently expand the path ahead. Moreover, the goal node is not known at start which also impossibilitates SRT’s applicability.

As stated on subsection 2.1.2, the control architecture developed in [7] is very interesting as it defines two control parameters groups: the velocity related attributes and the steering related attributes.

¹⁰This limit is also called the flicker fusion threshold, as described on https://paulbakaus.com/tutorials/performance/the-illusion-of-motion/ and https://en.wikipedia.org/wiki/Flicker_fusion_threshold (as consulted on Tuesday 12th September, 2017).
This division was considered for this work, as it revealed to be a good way of managing the control information.

The fact that other RSA like MCTS (although not GPU parallelized) proved to have positive results on TORCS also captivated the interest to develop this work. The evaluation function present in [8] (seen on equation (2.1)) served as a base to define this work’s prediction model and best state calculations, as it represents a good compromise between the distance travelled in a straight line and attenuation of that distance when driving on corners. The state representation present in [8] also inspired this implementation, as it uses physical aspects such as positions and velocities. A description of the state representation considered for this work can be viewed in section 4.2. Moreover, as stated on subsection 2.1.2, the evaluation methodology presented in [8] is very interesting as it used a typical TORCS qualification in-game time span (10000 game ticks) to compare the bots’ executions. Such time span was used in some of this work’s tests.

The benchmark mentioned in section 2.6 ($\approx 0.0417$ seconds per frame) was analysed while testing this approach, as it was considered as a target for the search times. Search times that did not deviate much from this benchmark were considered effective.

As mentioned when describing the work in [8], in TORCS some of the information concerning the car is hidden to the controller interface. For example, some aerodynamic and mechanical factors are not present. A PID controller was then used to compensate the errors achieved by forward model approximations. Reinforcing the point stated in subsection 2.1.3, the work in [14] served as a base for building and tweaking the pedals PID controller (referred on section 4.1) so that it responded adequately.
After analysing existing RRT parallelization approaches, it was concluded that a more suitable one was needed. Existing approaches were either too simple for this study [2] or applied to distributed contexts instead of in parallel ones [10]. This chapter describes a sub product of this work, which is a variant of RRT, named Iterative Parallel Sampling RRT. A description of the variant is followed by a presentation of its characteristics.

### 3.1 IPS-RRT Description

To create effective AI by the use of GPGPU, an RRT variant was created. It was named Iterative Parallel Sampling RRT (IPS-RRT), because instead of applying constraint parallelization (like in [2]) or creating parallel trees (like in [17]), it focuses on executing several iterations of parallel tree samples. At each iteration, a number of samples are concurrently generated and checked for constraints. The valid samples are synchronized to the global tree at the end of the iteration, which occurs when all threads have processed its sample. IPS-RRT’s pseudocode is presented in Algorithm 5.

**Algorithm 5 IPS-RRT**

1: `procedure GENERATE_RRT(numIterations, numParallelSamples, T)`
2:  `loop from 0 to numIterations:`
3:  `launch numParallelSamples threads computing this:`
4:  `T' ← T`
5:  `x_rand ← randomState()`
6:  `x_near ← nearestNeighbor(x_rand, T')`
7:  `x_new ← applyDelta(x_rand, x_near)`
8:  `if not validPoint(x_new) then`
9:  `endThread()`
10: `add vertex x_new to T'`
11: `add edge (x_near, x_new) to T'`
12: `synchronizeThreads(), T ← all T's`
13: `return T`

IPS-RRT receives three parameters: (1) a number of iterations, (2) the number of concurrent samples to be generated in each of those iterations and (3) an initial search tree $T$ that corresponds to the global tree. Partial trees (represented by $T'$) are used inside each thread. They consist in snapshots of the contents of $T$ at the start of each iteration (line 4).

A state $x_{rand}$ is randomly generated (line 5). Then, the algorithm searches for the closest state $x_{near}$ to $x_{rand}$ in the tree $T'$ (line 6). However, $x_{near}$ is not added directly to the tree. In order to expand the tree in a controlled way, a new state $x_{new}$ is generated by moving $x_{near}$ a small distance $delta$ in the direction of $x_{rand}$ (line 7). If the new state is not valid (does not verify the problem constraints), it is not added to $T'$ (nor to $T$) and the thread stops its execution (lines 8 and 9). Otherwise, the new state $x_{new}$ is added to $T'$ (line 10) along with a new edge between $x_{near}$ and $x_{new}$ (line 11). This edge
Figure 3.1: Schematic comparison between an 800 state search tree generated by the sequential RRT version and trees generated by IPS-RRT. The state space is projected to a 2D plane. Black points represent 2D states, black connections represent parent-child relations and the bigger red point is the initial state.

represents the parent/child connection between these two states. At the end of each iteration, threads are synchronized \(^1\) and the samples generated in each thread are copied over to \(T\) (line 12). Lastly, the algorithm returns the global tree \(T\) (line 13).

It is important to compare IPS-RRT with BSD-RRT [10] because it also parallelizes the creation of a single RRT tree. However, the process of building the trees is different, because of the way IPS-RRT’s schedule works. IPS-RRT iteratively builds partial trees by thread synchronization instead of incrementally broadcasting them in a distributed process context. In other words, IPS-RRT threads generate only one sample between synchronization points and a process in BSD-RRT generates several samples between a tree broadcast. IPS-RRT synchronization guarantees that updates to the global tree are seen in the next iteration. One alternative closer to BSD-RRT’s approach would be to update the global tree after a certain number of additions and propagate such changes to all threads, updating their partial graphs, but that would be inefficient in parallel contexts. Therefore, IPS-RRT process is more adequate for a GPGPU implementation. Another difference is that in BSD-RRT, a broadcast is done only after a number of new valid states are added, while in IPS-RRT synchronization occurs after each thread processes a sample independently of being added to the tree or not. The reason for this change is that since the synchronization must wait for the slowest thread, it would not be efficient to wait for an unlucky thread processing several invalid samples. Although IPS-RRT can return an empty tree when all states are culled, this problem can be attenuated by using a big number of samples, thus helping to cover a wider extent of the search space.

\(^1\)By thread synchronization it is meant that the threads wait for the slowest one before the execution continues.
3.2 IPS-RRT Characteristics

By selecting different number of iterations and number of parallel samples, IPS-RRT presents different characteristics. Fig. 3.1 illustrates how IPS-RRT trees can significantly differ from the classic RRT trees. Unlike the evenly distributed points seen in RRT trees, IPS-RRT trees reveal to have their points closer to the initial point. This fact is due to its schedule. Because of the synchronization of threads and the use of partial graphs, the trees are expanded in breadth rather than in depth. If one tries to maximize the parallelism by having just one iteration and performing all samples in parallel, then all resulting new states will connect to the initial state (given that each partial tree only has the initial state). This is illustrated in Fig. 3.1 b). By increasing the number of iterations, the algorithm will increase the span of the search tree and will be able to reach more distant states. When the number of iterations is equal to the number of generated samples (only one sample is created in each iteration), IPS-RRT becomes equivalent to the traditional RRT as illustrated in Fig. 3.1 f).

Based on the observation of Fig. 3.1 (and the experience gained with the work described here), some conclusions were taken about IPS-RRT. First of all, IPS-RRT is a simple and straightforward way of dividing the work done in the sequential RRT. Secondly, when the number of iterations is low, IPS-RRT is not applicable to problems where a distant solution needs to be calculated in the first search, because in such cases, the generated trees are small and compact. However, in problems where the solution can be obtained by determining smaller intermediate plans and executing them (as in the case of racing videogames like TORCS), the previous limitation is not as problematic and IPS-RRT becomes a good choice as a global solution can be obtained by a composition of solutions of smaller span plans.
This chapter presents the implementation of the developed CUDA accelerated TORCS bot using IPS-RRT. The solution architecture is presented and its components analysed. The state representation and restrictions are also detailed and depicted for better reader comprehension. A detailed schedule of this bot’s development is included in the appendix of this document, namely in part A (starting at page 59).

4.1 Bot Architecture

The developed TORCS bot consists of several modules. Those modules work almost independently from each other apart from some information exchanges. Because of such characteristic, this architecture is flexible and scalable. A schematic representation of the solution can be found in Fig. 4.1. It includes:

- **The TORCS back-end** which provides the necessary game core procedures to describe and control the car and query track information.
- **A planning module** which is composed by IPS-RRT ran on a CUDA kernel, as well as the plain RRT (for comparison purposes);
- **A control module** which receives a plan from the planning module and coordinates the actions needed to drive the car by calling the TORCS back-end. It uses a PID controller based on the plan information to define the commands passed to the back-end.

This architecture relies on a call cycle, which involves the planning and control modules and allows for a good coordination between layers. It is very important to understand this loop because it is fundamental to the good functioning of the bot. A scheme of the call cycle is presented in Fig. 4.2.

The planning module is called when the control module checks that either the car passed the last point of the current plan or a time limit is reached (defined by the attribute `SEARCH_RECALC_DELAY` and tweaked to 150 game ticks). When the planning module returns a new plan, the control module executes it. This procedure is repeated until the end of the race.

4.1.1 TORCS Back-end Parameters

The TORCS bots have some parameters that can be used to control the car. The main parameters are the car accelerator command (`car_accelCmd`), the brake command (`car_brakeCmd`), the gear command (`car_gearCmd`) and the steering command (`car_steerCmd`). While the accelerator and brake commands serve to define the position of the pedals, the gear command serves to define the current engaged gear and the steering command defines the angle of the car’s wheels in relation to the chassis.
Figure 4.1: Diagram of the TORCS bot’s architecture and modules.

Figure 4.2: Representation of the bot’s call cycle.
These variables define the information that is passed to the TORCS back-end and have to be considered when analysing other modules’ procedures. A description of the TORCS bots control parameters can be consulted in Table 4.1 and their function is clarified by the scheme provided in Fig. 4.5.

Table 4.1: TORCS Back-end Control Parameters

<table>
<thead>
<tr>
<th>Var</th>
<th>Min Value</th>
<th>Max Value</th>
<th>TORCS API Attr.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gas Pedal Command (%)</td>
<td>0 (0%)</td>
<td>1 (100%)</td>
<td>car._accelCmd</td>
</tr>
<tr>
<td>Brake Pedal Command (%)</td>
<td>0 (0%)</td>
<td>1 (100%)</td>
<td>car._brakeCmd</td>
</tr>
<tr>
<td>Gear Selection Command</td>
<td>0</td>
<td>(depends on the car)</td>
<td>car._gearCmd</td>
</tr>
<tr>
<td>Steering Command (radians)</td>
<td>0</td>
<td>car._steerLock</td>
<td>car._steerCmd</td>
</tr>
</tbody>
</table>

4.1.2 Planning Module

The planning module consists of an abstraction for the execution of both the plain RRT and IPS-RRT. The programmer decides which search algorithm is used in a certain bot execution. This module implements an interface which can be used to represent both versions. Each planning algorithm implements a method named `search()`, which computes a whole search and returns the best plan. Besides that, both versions also implement a method called `getGraph()` which returns the currently generated tree. The output of the planning module is the best plan without the initial point (the initial point represents the current state which is redundant to seek).

A big advantage of the planning module is that it can be extended to have more search algorithms (if needed for further comparison) as it abstracts the planning process. The only aspect that has to be accounted for is that all search algorithms have to implement the same interface. Such interface is included in this document’s appendix, namely in part D (starting at page 66).

4.1.3 Control Module

The module that controls the car is divided into three components: the pedals component, the steering component (similarly to the speed and pose controllers described in [16]) and the gear component. It drives the car using the TORCS control parameters depicted in Fig. 4.5 and presented in subsection 4.1.1. The control module is detailed on Figure 4.3.

The pedals are controlled by a PID controller [14], as the relation between their position and the car’s acceleration (and velocity) varies from car to car and cannot be directly acquired. The output of the PID controller is a number between 1 and -1. If the output is positive, a gas command is called (`car._accelCmd = output`); else, a brake command is called (`car._brakeCmd = -1*output`). The objective of using PID is to minimize the errors incurred by the search, getting the predicted and experimental velocities close to each other using the pedals. The use of this type of controller revealed to be a good compromise. The used PID parameters for this component were $K_p = 0.008$, $K_i = 0.005$ and $K_d = 0.001$. As the application was asphalt racing, the use of small $K$ values (topic approached in [14]) proved to be effective.

A PID controller was not needed for steering given that the steering angle can be determined as represented in equations (4.1) and depicted in Fig. 4.4. The angle between the car’s direction and the current state position is obtained by the arc-tangent of the legs of the triangle (by calling the method `getSteer(<targetAngle>)`). In this case, the calculated information is not an approximation and is only limited to the minimum and maximum steering angles (`car._steerLock`). Although allowing for abrupt
steering changes, which are not realistic, the simulation permits such behaviour. Besides that, because a physical based forward model is used to produce the plans (as detailed on section 4.3), it is ensured that no abrupt steering changes are really considered.

$$\text{targetAngle} = \text{arcTan}(\text{target.y} - \text{car.pos.Y}, \text{target.x} - \text{car.pos.X});$$  

$$\text{targetAngle} = \text{car.yaw};$$  

$$\text{targetAngle} \in [-\pi, \pi];$$  

$$\text{targetAngle} = \frac{\text{targetAngle}}{\text{car.steerLock}};$$

Finally, the gear selection (method `getGear()`) is based in the simple “bt” TORCS bot. It uses the engine’s rotations per minute (`car.engineRpmRedLine`) to check if a gear change is needed. More specifically, this method considers an engine rotation percentage limit and calculates if the speed that can be reached with the current gear is adequate or not. If the car is going too fast or too slow for the current gear, it sends an up or down shift commands (`car.gearCmd = car.gear + 1` or `car.gearCmd = car.gear - 1` respectively). If the speed reveals to be adequate for the current gear, there is no gear shift (`car.gearCmd = car.gear`).

### 4.2 TORCS Search State Representation

The objective of each state is to represent a predicted car situation. Thus, the representation that seemed more adequate was including attributes like position and velocity. In TORCS, the track maps include lower and higher absolute position bounds, along with bounding boxes of segments that represent the limits of the track (such segments are depicted on Fig. 4.8). Therefore, the position of the car is represented in the cartesian space defined by the TORCS back-end and the velocity is represented in a polar space aligned to the TORCS back-end space. In this implementation, the velocity is represented in polar space, which means it has an angle and intensity. While the intensity is bound to
the car’s top speed or can be tweaked in order to limit the car’s speed on a specific track, the angle is bound between $-\pi$ and $\pi$ (the full angle range). A polar space was chosen for the state’s representation of velocity because it is simple to visualize in a racing context, as it separates the velocities’ intensities (related to pedal control) and their angles (related to steering control). The downside is that the TORCS back-end represents velocities in cartesian space. Therefore, the velocities have to be converted to a cartesian space or polar space before being passed to or after being extracted from the back-end. Such conversions are presented in equations (4.2).

Considering a velocity in $(x, y)$ to convert to $(\text{intensity}, \text{angle})$:

\begin{align*}
\text{intensity} &= \sqrt{x^2 + y^2} \\
\text{angle} &= \tan^{-1} \frac{y}{x}
\end{align*}

Considering a velocity in $(\text{intensity}, \text{angle})$ to convert to $(x, y)$:

\begin{align*}
x &= \text{intensity} \times \cos(\text{angle}) \\
y &= \text{intensity} \times \sin(\text{angle})
\end{align*}

Besides the TORCS control parameters mentioned in subsection 4.1.1, the function of each state attribute is also clarified by the scheme provided in Fig. 4.5.

4.3 RRT and IPS-RRT Applied to TORCS

As described in section 4.1, the planning module includes a sequential and parallel versions of the search algorithm. Both algorithms represent the search tree as an array of states. It is important to point that each state’s input is not calculated on demand like presented on Algorithm 1 (represented by the variable $u$). Such input is inferred afterwards by the control module (more specifically by the control components such as the pedals PID controller) instead of in the planning phase. It is also important to notice that there is an implementation difference between IPS-RRT applied to TORCS and the pseudocode presented on section 3.1: threads do not keep copies of the main tree, as it reveals not to be efficient nor practical. Instead, the neighbours are picked from a constant memory tree, which is updated at the start of each iteration. This ensures that although some threads can see the global
memory tree updated by other threads, they will only write to such global tree. The threads always read from the constant memory tree copy, which is, as mentioned before, a snapshot of the global tree at the start of each iteration. As a result, the trees obtained by this implementation are similar to the ones presented in Fig. 3.1. The presented implementation detail adapts to the CUDA paradigm as global and constant memory management is easy and straightforward: a global tree simply had to be created, copied to constant memory and passed to the search kernel. Care was taken not to make global memory writes create race conditions between threads: it is guaranteed that each thread updates a different position of the global tree. The tree is initialized with as much positions as number of samples to be generated (each position is initialized with a special flag), plus one position for the initial node. If a sample cannot be inserted (because it was pruned, for example), it will not be considered every time the global tree is traversed, as the position of the tree that would have that sample still has the initialization flag. Fig. 4.6 depicts the tree representation used on the IPS-RRT CUDA kernel.

Next, the implementation of the IPS-RRT methods presented on page 21 (which include the RRT methods described on page 8) is detailed. First, a velocity is randomly sampled and used to generate a new state $x_{rand}$ (line 5 of IPS-RRT’s pseudocode). As the state space is formed based on velocities, the states’ positions have to be computed afterwards. Therefore, a forward simulation method considers a simple physical model to generate the position of $x_{rand}$ based on its velocity. The position is obtained by multiplying the velocity by a fixed search action time. Such time has to be carefully tweaked for the characteristics of the track being used. If it is too big the forward model thinks the car is going very
fast and brakes often. Oppositely, if such time is very small, the forward model thinks the car is going very slowly and tends to overshoot corners by going too fast. Nevertheless, by only using velocities and a fixed time span, the forward model becomes very simple. This is beneficial for contexts such as in GPGPU because (as already noted in section 2.5) GPU cores are most efficient when processing simple information.

The nearest neighbour $x_{\text{near}}$ (line 6) is determined by choosing the state with the lowest velocity angle variation to $x_{\text{rand}}$ (the angle variation between two velocity vectors is the minimal angle between them). This computation is simple but revealed to be enough to produce adequate results. The intensities were not included as it would be hard to balance the cost of the angles and intensities correctly (there is no relation between the values of these attributes). Moreover, the intensities error is amortized by the pedals PID controller and so a bigger intensity discrepancy can be tolerated. Such discrepancy is also reduced by using a delta to compute new states (represented by $x_{\text{new}}$ at line 7), thus limiting the states variations. To calculate $x_{\text{new}}$, a state is picked from a set of intermediate states generated between $x_{\text{near}}$ and $x_{\text{rand}}$. Such states are computed by interpolating their velocities. Such interpolation is divided in two steps. Firstly, the angles are linearly interpolated. Secondly, the intensities are interpolated using the equation (4.3), where $t$ is the interpolation percentage between 0 and 1, $a$ is a tweaking parameter and $p_i$ is the intermediate point between $p_0$ and $p_1$. The value used for $a$ in this implementation was 0.03. When using such value, the intensity interpolation varies less as the difference gets higher and can better approximate the telemetry of the car used for the evaluation phase. Fig. 4.7 illustrates the states interpolation procedure. While executing the bot, 20 intermediate states were generated and the 5th state was picked as $x_{\text{new}}$.

$$t = \frac{t}{\ t + a}; p_i = p_0 * (1 - t) + p_1 * t$$

Finally, $x_{\text{new}}$’s validity (lines 8 and 9) is checked by verifying if the intermediate states are positioned inside the track. The track is divided into segments that represent its drivable boundaries. A representation of the segments that compose the track is presented on Fig. 4.8. The track segments data model used for this implementation is different than the one provided by the TORCS back-end. Instead of directly using the linked list available on the bot’s interface, it was converted to an array in track loading.
Intermediate states and \textit{delta} representation. \textit{x}_{\text{rand}} is represented in dark blue and \textit{x}_{\text{near}} in light blue. The states’ velocities are also depicted accordingly. Intermediate states are represented in grey. \textit{x}_{\text{new}}, represented in orange, is determined by choosing one of the intermediate states. In this case, the third furthest state from \textit{x}_{\text{near}} was chosen. However, the chosen intermediate state is defined initially, just like a \textit{delta} in RRT.

Representation of the segments that compose the track. The track segments bounds can be identified in orange. The segment occupied by the car is coloured red. The others are coloured blue.

time. This way, the segments could be passed to the search CUDA kernel, as it is easy to copy contiguous memory spaces to the GPU. To test if a state is inside the track, the segments array is traversed and each of its four bounds compared with the state’s position coordinates. If a segment containing the state is found (the state’s position coordinates lie inside one of the segment's bounds), the state lies on track. The calculation to check if a state lies inside one segment is depicted on Fig. 4.9.

After the search tree is built and returned (line 13), the best tree state is picked as the one estimated to maximize the covered track distance in the minimum amount of time. The track distance for a state \textit{s} is determined by projecting the line segment between the initial state and \textit{s} along the centre of the track, and calculating the length of the projection. The path cost is then calculated by the ratio between the track distance and the depth of the state in the tree. Since a tree connection represents a fixed time span, the depth of the state in the tree can be used as an estimation of the time required to reach that state. In the end, the best state is backtracked to the initial state and the corresponding path is returned as the solution.

\subsection{Implementation of a Sequential version of IPS-RRT on the CPU}

For comparison purposes, a CPU version of IPS-RRT was developed. It runs the IPS-RRT’s iterations sequentially, copying the data to partial trees and only updating a global tree after a certain number
Figure 4.9: Calculation to check if a state lies in one segment. The distances between the sides and the position of the state (represented in blue) are checked to be lower than the distances between the extremes of the track’s segment (represented in brown). If they are, the state is considered to be inside the segment. In this case, the state with position $P_{state}$ lies inside the segment $[P_0, P_1, P_2, P_3]$, because (1) the distances between $P_{state}$ and either $P_{(0,2)}$ or $P_{(1,3)}$ are lower than the distance between $P_0$ and $P_1$, and (2) the distances between $P_{state}$ and either $P_{(0,1)}$ or $P_{(2,3)}$ are lower than the distance between $P_1$ and $P_3$. Although this consists of an approximation, the error is negligible as segments are nearly rectangles (given that TORCS tracks generally augment the segment sampling in sharp turns).

Figure 4.10: A representation of the line formed by two states on the bezier forward model. The states’ positions are represented in blue and their velocities in red. The parent is the state on the left and the newly generated state the one on the right. The inverse of the newly generated state’s velocity is represented in pink.

of iterations. This version was useful to see the impact that IPS-RRT’s scheduling had on the search and lap times, or in other words, in what aspects those parameters changed without taking in account the parallelization influence. This version is included when presenting and analysing the tests of subsection 6.3.2 and section 6.4.

4.3.2 Other Tested Forward Model

Another implementation of the forward model was considered before the one described in section 4.3. Although managing to drive the car around parts of some tracks, it revealed to be very inconsistent, getting the car off track a big number of times. It modelled the trajectories of the car by applying a bezier line approximation. The curves control points included not only the newly generated states’ and their parents’ positions, but also the parents’ positions shifted by their velocities and the states’ positions shifted by the inverse of their velocities. An example of a produced bezier line is displayed in Fig. 4.10.

The delta positions were calculated (in applyDelta) by limiting the distance between the newly gen-
erated states’ and their parents’ positions along the bezier. The delta velocity was found by calculating the derivative of the bezier curve in the delta position.

The big drawback of this model is that it tried to represent a physical aspect (the car’s trajectories) using a mathematical concept. The model did not work correctly because it did not consider that when the car goes faster, its trajectories tend to straighten up, presenting different topologies than trajectories followed at low speed. Therefore, the paths constructed by this technique were often jagged and inconsistent (in what respects not only to the positions, but also the velocities), by which it was determined that a more exact forward model needed to be developed. This need led to the development of the physical model presented in the beginning of this section.

4.4 Writing Statistics to a Log

To ease the attainment of the search and lap times, a log writer was created. It incorporates two methods: the writeToLog and closeLog. The first method creates (if not already created) and writes a new line to a log file (it receives the file name). A line is written everytime a plan is returned. Such line contains the time spent on that search (time span between the call and return of search). Everytime the car finishes a lap, the lap time is recorded. It is obtained through the TORCS bot interface by checking the attribute car.race.lastLapTime. Both search and lap times are recorded in seconds. The resulting log is a csv file which can be used to create charts. An excerpt of the log generated by a parallel search instance is depicted on Listing 4.1.

Listing 4.1: Example of the generated log file

| searchID ; numStates ; numIterations ; run ; searchTimes ; lapTimes |
|-----------------|-----------------|-----------------|-----------------|-----------------|
| ParallelRRT ; 800;4;2;0.034000 |
| ParallelRRT ; 800;4;2;0.034000 |
| ParallelRRT ; 800;4;2;0.035000 |
| ParallelRRT ; 800;4;2;0.034000 |
| ParallelRRT ; 800;4;2;0.034000 |
| ParallelRRT ; 800;4;2;0.032000;40.452002 |
| ParallelRRT ; 800;4;2;0.034000 |
| ParallelRRT ; 800;4;2;0.037000 |
| ParallelRRT ; 800;4;2;0.034000 |
| ParallelRRT ; 800;4;2;0.034000 |
| ParallelRRT ; 800;4;2;0.033000 |

The resulting csv file contains the following attributes: (1) searchID which identifies the type of search, (2) the number of generated search states numStates, (3) the number of iterations numIterations (if using IPS-RRT), (4) the test run and most importantly (5) the search times searchTimes and (6) the lap times lapTimes. By grouping these values by searchID, numStates, numIterations and run and finding the mean, minimum or standard deviation of the searchTimes and lapTimes, charts were created and used to evaluate this work (such charts are displayed on chapter 6). In order to manipulate the data that was recovered from each test, a program called Pentaho Data Integration was used. A transformation was built in order to extract and transform the data correctly.

\[\text{\textsuperscript{2}}\text{http://community.pentaho.com/projects/data-integration/ (as consulted on Tuesday 12th September, 2017).}\]
This chapter presents some preliminary tests which were conducted throughout the development of this work. As the GPGPU code revealed big inefficiency, some optimizations had to be done. Such optimizations are also detailed.

During all tests, the used graphics card was an Nvidia GTX 960M (laptop) GPU with 4GB of dedicated VRAM. It has a 5.0 compute capability and a 1.176 Ghz clock rate. The device copy overlap and kernel execution timeout flags were enabled during the execution of the bot. This GPU has 5 multiprocessors with 65536 registers per multiprocessor and 32 thread warps. The maximum number of threads per block is 1024 and the maximum thread dimensions are (1024,1024,64). The constant memory size is approximately 65 KB and the shared memory size per multiprocessor (managed cache) is approximately 49 KB.

The used CPU was an Intel(R) Core(TM) i7-4720HQ (laptop) CPU with a 2.6GHz clock (3.6GHz of maximum turbo frequency) and 6 MB of cache. This CPU has 4 cores.

In what respects to the software, the work was developed in the C++ programming language, using Microsoft’s Visual Studio\(^1\) 2013 and CUDA development toolkit\(^2\) 7.5. This combination was chosen because despite more recent ones being available, it was the most stable and well documented when the development started.

## 5.1 Preliminary Tests

In this subsection, some preliminary solutions are described. They consisted not only in an implementation developed alongside the proposal for this work, but also a test which allowed to extract some important conclusions about this approach. The latter involved the first full implementation of the bot, which was not optimized and presented some problems later on.

### 5.1.1 Algorithm Test / Proof of Concept

An implementation of the RRT was tested to check the adaptability of the RRT family to the context described in this work. The objective of this solution was to demonstrate the feasibility of the work, or in other words, to serve as a proof of concept. Using simple states to represent predictions, which contained positions, velocities and accelerations, and limiting the tree span to 30 segments ahead of the car, random trees were successfully built and plans were generated. The attribute acceleration was included because it could be considered in the forward model later on, but it was not needed in the final solution. A new search calculation was called every time the car reached the last point of the current

---

\(^1\)https://www.visualstudio.com/ (as consulted on Tuesday 12\(^{th}\) September, 2017).

Figure 5.1: Chart displaying the mean search times for the parallelization test. The bottom part includes the used search parametrizations and the label presents the number of generated states. For example, in an 8 IPS-RRT Iteration search generating 1600 states there are 200 parallel threads (1600/8).

search. The used evaluation function was the quadrance of the euclidean distance between points. The formula can be seen in equation (5.1).

\[(x_{parent} - x_{curr})^2 + (y_{parent} - y_{curr})^2\] (5.1)

The plan path was found by backtracking the initial point's furthest state (like in the final approach). The bot followed such path, state by state. In this test, the control module was simpler as it considered fixed percentages for the pressure on the pedals. However, the steering submodule was almost identical to the final one. Although the bot's behaviour was inadequate due to poor control (too much cornering speed) and even casually random (as expected from a stochastic algorithm in which the forward model is not fully developed), the solution inspired a skeleton for the work here presented as its architecture was easily extended to produce the final one.

5.1.2 Parallelization / First Complete Implementation Test

After developing the first version of IPS-RRT, some tests were conducted to test the applicability of such parallelization method to TORCS. The mean search and lap times acquired from the execution of several searches during 3 laps on the “CG Speedway number 1” track can be seen in the charts of Figs. 5.1 and 5.2.

In Fig. 5.2, the acronym “NC” (Not Consistent) means that the search executed some recoverable mistakes (the data was measured on three non-consecutive clean laps). A recoverable mistake means that the car gets out of track but can return by itself. The acronym “DNF” (Did Not Finish) means that the search failed to finish a lap, committing an unrecoverable mistake (like when the car gets out of track and the search span is not big enough to bring it back).

Although it seemed that the GPU implementation was always faster than the CPU version, after some analysis were made to the CPU version along with some improvements to the algorithm procedures, it was concluded that this is not the case. When some optimizations were done to TORCS compilation, the sequential RRT times lowered significantly. Chapter 6 presents a set of tests which,

---

3The quadrance of a distance was used instead of the actual distance because it is more efficient to compute (a square root is a demanding operation) and it still ensures a correct result when comparing values. For example if \(a > b\), then it is guaranteed that \(a^2 > b^2\).
besides approaching the topics underlined in section 1.1, also help the reader comprehend in what situations can the application of IPS-RRT to the GPU really be useful. Nevertheless, several conclusions were taken from the execution of this preliminary test, as some problems inherent to this approach were detected right away. Such problems are represented in Fig. 5.3. They consist in:

- When using a number of IPS-RRT iterations lower than 4, the algorithm was not able to search far enough to detect incoming corners (Fig. 5.3 a);

- When using a big number of tree samples, the search took so long that the returned solution was no longer consistent with the car’s current position (Fig. 5.3 b), and the lap times attenuated. This happens because the TORCS simulation is frame independent,

By using the Visual Studio Performance Wizard with CPU sampling on each of this version’s sequential method calls, it was confirmed that the time was spent mostly on constraint checking (13.63% of the samples). The random state generation and nearest neighbour checking procedures were much faster (they both took only a sum of 0.21% of the samples). This result corroborated the conclusions mentioned in [2], as constraint checking seemed to also be the most computationally expensive method in that work.

---

Table 5.1: Search Times (in seconds) executing an 800 states 1 Thread IPS-RRT search on the GPU (initial version)

<table>
<thead>
<tr>
<th>Step Description</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Just the kernel (nothing on it)</td>
<td>0.04</td>
</tr>
<tr>
<td>After adding random state generation</td>
<td>0.05</td>
</tr>
<tr>
<td>After adding nearest neighbour calculations</td>
<td>1.93</td>
</tr>
<tr>
<td>After adding delta appliance</td>
<td>1.94</td>
</tr>
<tr>
<td>After adding constraint checking</td>
<td>4.95 to 6.07</td>
</tr>
</tbody>
</table>

5.2 CUDA Implementation Optimizations

As presented above, after executing the preliminary tests, some optimizations were made to the TORCS compilation, which vastly improved the sequential RRT’s efficiency. However, it was then observed that the CUDA implementation was executing very slowly, getting much bigger search times than the CPU, even while running thousands of threads. In order to decrease search times, the code was profiled and carefully analysed. Such analysis consisted in running an IPS-RRT 800 state search in the GPU with only one thread and comparing it with the sequential RRT, while running both searches in the same track and with the same forward model parametrizations. While the CPU was executing such test in a few centiseconds, the GPU was taking around 5 to 6 seconds. Research was done to understand what was the cause of such performance discrepancy. In the end, it was concluded that the differences in memory latencies were majorly responsible for such phenomena. The CPU has a large cache and can copy all the contents of the tree and segments arrays to it. Therefore memory accesses have low latency (a CPU like the one used in this work has an L2 cache latency of 12 cycles, which corresponds to $\approx 4.62 \times 10^{-9}$ seconds)\(^5\). Oppositely, accesses to the VRAM are costly: according to [1], the global memory latency is about 300 cycles (in the used hardware, this value corresponds to $2.55 \times 10^{-7}$ seconds), which is orders of magnitude higher than any of the presented latencies. As described next, some fixes were implemented, achieving a decrease of around five times in the CUDA implementation execution time.

In order to analyse where the performance was lost, the kernel was executed 5-10 times with and without some parts of the algorithm. It was observed that the most significant loss in efficiency came not only from constraint checking, but also from the nearest neighbour procedures. Moreover, as introduced before, it was concluded that the memory accesses were responsible for the discrepancy in search times in the CPU and GPU, as the accesses done on these methods by the GPU implementation always implied going to the VRAM. As described in 2.5.1.2, the VRAM is located outside the multiprocessors, which means it has high latency. The cache however is located on chip, and by transferring data to it (directly by shared memory or indirectly by constant memory) the efficiency of the GPU implementation could be improved (according to [1], the GPU’s cache memories have latencies of only about 11 processor cycles). A summary of the data recovered from this test is shown in Table 5.1.

First, constraint checking was optimized to reduce the number of memory accesses. The search for the current state’s segment started at the beginning of the segment list. Instead of starting the traversal at the beginning, it was changed to start at the array position of the parent’s segment. Such traversal then stops when the segment which contains the tested state is found. This heuristic assures that segments which are located further from the start of the segments array are more easily found, as the neighbours segments are generally close to each other. This increased data locality results in fewer tested segments, and consequently better performance. After applying this improvement, the execution

\(^5\)Moreover, a RAM access takes about 36 cycles ($\approx 1.38 \times 10^{-8}$ seconds).
Table 5.2: Search Times (in seconds) executing an 800 states 1 Thread IPS-RRT search on the GPU (after some optimizations)

<table>
<thead>
<tr>
<th>Description</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Just the kernel (nothing on it)</td>
<td>0.04</td>
</tr>
<tr>
<td>After adding random state generation</td>
<td>0.05</td>
</tr>
<tr>
<td>After adding nearest neighbour calculations</td>
<td>0.33</td>
</tr>
<tr>
<td>After adding delta appliance and constraint checking</td>
<td>0.93</td>
</tr>
</tbody>
</table>

time of the test presented above (relative to the GPU implementation running one thread) was decreased to about 2.5 to 2.7 seconds. The change in the CPU time was noticed too, although in a different scale (a few milliseconds).

Another aspect which helped to lower the search time was removing some unused states' attributes and using simpler segment structures than the ones delivered by the TORCS API. After such fixes, the kernel was executed one more time, with and without some parts of the algorithm. Just by doing this, the search time was reduced to about 1 second. A summary of the data recovered from this test is shown in Table 5.2.

After analysing Table 5.2, it was concluded that some improvements could also be implemented in the nearest neighbour calculation. As described in 2.5.1.2, CUDA's programming model presents three main types of memory, the global memory (per grid memory), a per block shared memory and a per thread local memory. By far, the global memory is the most costly of the three, because it not only has the biggest latency, but also copies from main memory to it/ from it to main memory (RAM to the graphics card VRAM/ VRAM to RAM) can be slow. As a full tree traversal was needed on nearestNeighbor, the only thing to do was to pass the tree information to a lower latency memory. However care was taken to make a lightweight version of the tree which only contained velocities. Such structure could then be fitted to shared or constant memories, which have limited size.

Initially, shared memory was used to accommodate the lightweight tree. One disadvantage of using shared memory was that a thread had to copy the full tree from global memory everytime a kernel was called. As described in 2.5.1.2, shared memory is not persistent (it represents the cache of the GPU) and therefore does not keep its information between calls. Another concern was that this memory is only accessible per block. Nonetheless, it reduced the search time to approximately 0.8 seconds, because of its lower latency. After analysing the disadvantages of using shared memory, effort was made to find a more suitable memory to save the tree.

Such research led to the use of constant memory. Constant memory resides at the device memory (as depicted on Fig. 2.8), but as mentioned in 2.5.1.2, it has a particularity: it is cached. This means that the latency of transferring data from/to the device memory can be avoided everytime there is a cache hit. This memory seemed to be more adequate for this case, as it has best performance when applied to algorithms which have a high number of accesses to the same data. Tree accesses in nearestNeighbour are of such type: the tree is concurrently accessed by all threads. As constant memory is read-only, the tree is copied from global to constant memory at the start of each iteration. As this copy is made on the GPU itself, it is fast and does not represent a bottleneck. After implementing this optimization, the search time was reduced to about 0.7 seconds.

As referred before, constant memory can only contain constant symbols and shared memory is not persistent. Therefore, a full version of the tree was still maintained alongside its lightweight version in order to accommodate global tree changes and update non persistent memories between IPS-RRT iterations.
Texture memory was not considered because although it could improve the efficiency of some calculations like interpolations, it works the same way than the constant memory, and so its latency is also dependent on the cached data. Moreover, memory accesses in nearestNeighbor and constraintChecking are traversals to the tree and segments lists which do not represent random accesses.

Copies from RAM to the graphics card global memory and from global memory back to RAM are slow. Such copies could also be minimized in order to increase the CUDA algorithm performance. Therefore, the tree and segments arrays were copied from RAM to global memory in track loading time, instead of executing that copy when the search was called. A method named gpuInit was used before the main search kernel to perform such operation. When a search was called, the only thing that had to be done was making sure to re-initialize the tree in order to clean previous data. A kernel named graphInit was created to do just that. Unfortunately, this optimization did not significantly improve the search time as the amount of copied information while running this test was not big enough to make a difference.

5.2.1 CUDA Warmup Kernel

A special kernel, called the warmup kernel, was included and ran in track loading time. This kernel is very simple as it is just a variable-value attribution. Its implementation was considered because the first CUDA instruction leads to the automatic creation of a context and as a result is slower than posterior ones. As cited on [1], “A CUDA context is analogous to a CPU process. All resources and actions performed within the driver API are encapsulated inside a CUDA context, and the system automatically cleans up these resources when the context is destroyed”. Having this warmup kernel helps to avoid extra delays when starting a race. The effectiveness of this kernel depends mostly on the information processed in the main kernels, as less demanding kernels that take little time to process have generally higher gains. Although not influencing the evaluation results, it was included to avoid the context creation delay in the bot’s initial search execution.

6The CUDA fact pointed on this part among others are included in https://cudaspace.wordpress.com/2013/04/04/difference-on-creating-a-cuda-context-for-timing-a-kernel-warmups/ (as consulted on Tuesday 12th September, 2017).
This section presents and discusses the results obtained by the execution of several tests aimed at studying the computational efficiency, the quality of the returned solutions, and the interaction between the graphics rendering and the search algorithm in the GPU.

The goal of the work is to design, explore and study the use of GPGPU-based parallelization techniques to improve the application of RRT to racing videogames. A set of aspects to be studied were underlined in section 1.1. The evaluation phase then tried to approach each aspect clearly, by:

- Checking the impact of using the same GPU to render the graphics and process the searches; Observing if the same GPU could effectively process the two aspects simultaneously;
- Testing the computational efficiency of the used parallelization method (IPS-RRT) by not only executing it with different parameters, but also executing the implemented bot for several runs in several test tracks while measuring the mean search times;
- Checking the quality of the returned solutions by executing the implemented bot for several runs in several test tracks while measuring the minimum and mean lap times.

Care was taken to choose tracks which eased the application of the implemented forward model, as it had to be tuned for each track (namely by altering the forward model action time, as explained on subsection 4.3). Screenshots of the bot’s execution in some of the test tracks can be seen in Fig. 6.1.

As presented in 2.5.1.2, CUDA threads are directly bound to the GPU multiprocessors and blocks are abstractions which represent groups of threads. In order to get the best results, the number of threads per block should be a round multiple of the warp size (32 threads on current GPUs). This way, the best possible data throughput can be achieved. Moreover, the number of blocks must be low enough to prevent them from executing sequentially (as depicted on Fig. 2.5). When running the GPU implementation of IPS-RRT in each test case, this factor was taken in account.

The used software and hardware were identical to the ones used in the preliminary tests and presented at the beginning of chapter 5.

Figure 6.1: Screenshots of the bot’s execution in some of the test tracks.
6.1 The Test Tracks

In this subsection, the used test tracks are presented. As mentioned in the start of the chapter, these tracks were chosen because they eased the application of the implemented forward model. Firstly, the track “CG Speedway number 1” is detailed. A representation of its layout is depicted on Fig. 6.2. It was chosen because it not only presents some bends that require some care not to overshoot, but also contains a big straight section which can attenuate the loss of time in such bends. The track has 301 segments. The in-game width of the track is 15 meters\(^1\).

The track “E-Track 5” was chosen because it has some left and right wide turns. Here, the forward model could be configured to really test the cornering limits of the car while having a low probability of generating empty plans. A representation of “E-Track 5” is depicted on Fig. 6.3. Such risky configuration lowered the overall consistency of the forward model, not because the search returned empty plans, but because sometimes the car ended up being too greedy, putting a wheel on the grass which made it spin. The track has 335 segments. The in-game width of the track is 20 meters\(^1\).

The track “Michigan Speedway” (its layout is presented on Fig. 6.4) was chosen because it is an easy track to achieve consistency, as its layout is nearly oval. Moreover it is a fairly small track, which allowed the bot to do a big number of laps during the tests. The track has 248 segments. The in-game width of the track is 18 meters\(^1\).

6.2 Studying the Graphics / Search Interaction

The first test consisted on executing 40 searches at the start position of the track “CG Speedway number 1” and checking the mean search times. The purpose was to compare the search and graphics execution in separate GPUs to the execution of both aspects on the same board GPU, checking the

\(^{1}\)In TORCS the tracks have a constant width. This is an aspect the game developers were hoping to later change while releasing the used TORCS build.
Figure 6.3: Layout of the track “E-Track 5”.

Figure 6.4: Layout of the track “Michigan Speedway”.
results of the interaction between the graphics rendering and search processing. The used search parametrizations were 2, 4 and 8 iterations of 800, 1600, 3200 and 6400 states. The results for 800 and 6400 states (the extremes) are displayed in the bar charts of Figs. 6.5 and 6.6. The full test is included at the appendix of this document, namely in part B (starting at page 61).

After analysing the charts, it can be seen that the search times are statistically the same, as there are no consistent search time increases or decreases when using different boards or the same board to execute the graphics and AI calculations. Moreover, no significantly negative impact was seen on the graphics execution. This leads to believe that the GPU driver’s scheduler is good enough to cope with the simultaneous execution of graphics and CUDA kernels. As this work focuses on the simultaneous execution of those aspects, the rest of the tests were conducted using the same board for the execution of both the graphics and searches.

6.3 Studying the Computational Efficiency

In order to approach this topic, a wide set of tests was conducted. The purpose was to (1) check the properties of IPS-RRT while being applied to this context by either augmenting the number of par-
allel iterations, maintaining the same number of states; or oppositely, increasing the number of states while maintaining the same number of parallel iterations and (2) after analysing the results of the tests mentioned in (1), checking which impact the GPGPU-based approach had in search times when more exhaustive tests were executed.

6.3.1 IPS-RRT Scalability Tests

These tests consisted in executing several groups of 40 searches at the start position of the track “CG Speedway number 1”, checking the mean search times while using different parametrizations. They display how IPS-RRT behaves as the number of iterations gets higher for the same number of states or oppositely, as the number of states increases with the same number of parallel iterations.

These tests provided some knowledge about this approach which was useful when analysing the tests presented later on. The search times seen on more exhaustive tests shared characteristics with these smaller tests.

6.3.1.1 Number of Iterations Test

In this test, each search generated 800 states, applying an increasing number of iterations, from 1 (800 parallel threads) to 800 (1 thread). This test was used to show how the search times grow when a higher number of parallel iterations is applied. 800 states were chosen because they are enough to reveal the expected proprieties, without being too slow when a search executes in only one thread (800 iterations).

As can be seen from the analysis of the scatterplot of Fig. 6.7, there is a sub-linear increase of search times as the number of iterations gets higher. Because of such increase, a balance has to be found between the reduction of search times and the depth of the plans. If a big number of iterations is used, the plans become deeper but the search time gets higher. Oppositely, when using a low number of iterations, the search times are low, but the plans’ depth is also low. In the meantime, it can be inferred that beyond 4 iterations, the search times escalate above the perception threshold ($\approx 0.0417$ seconds) using only 800 state searches. Nevertheless, in the preliminary test presented in subsection 5.1.2, 4 iterations revealed to be enough to produce consistent plans for “CG Speedway number 1”, which has a narrow road and therefore a limited search space.
6.3.1.2 Number of States Test

In this test, each IPS-RRT search generated an increasing number of states, from 100 to 6400 (chart of Fig. 6.8). For comparison purposes, the same test was also run for the Sequential implementation (chart of Fig. 6.9). This test was used to show how the search times are increased when a bigger number of states is generated.

The chart of Fig. 6.8 reveals that the search times tend to evolve sub-linearly as more states are applied. This can be explained because of the gain inherent to the parallelism. Oppositely, the sequential version (chart of Fig. 6.9) appears to scale much faster as expected. This test shows in particular that 4 iterations searches seem to maintain low search times. Moreover, 4 iteration searches revealed good bot performance in preliminary tests. Because of that, this number of iterations was chosen in subsequent tests.

In the end, it can be predicted that the bigger the problem size, the less impact adding states has on the search times of IPS-RRT, increasing its efficiency compared to the sequential version. However, as the search times increase, care must be taken not to deviate much from the perceivable threshold ($\approx 0.0417$), because big search times produce big drops on the frame refresh rate (frames displayed per
second), especially when the searches are fully executed on a single frame such as in this work.

### 6.3.2 Exhaustive Computational Efficiency Tests

This group of tests consisted in measuring the mean of search times for both the Sequential RRT and IPS-RRT while executing the bot for several runs of 10000 ticks (roughly 3 minutes and 26 seconds of in-game time), a typical TORCS competition qualification. As the search time decreases, it is assumed that the search is more efficient. The bot was executed for 10 runs on the tracks “CG Speedway number 1” and “Michigan Speedway”, as all searches were consistent when using a number of generated states equal or higher to 800 (remembering the definition of consistency from the preliminary test of subsection 5.1.2). Besides such tracks, the bot was also executed on “E-Track 5”. However, the searches on such track were less consistent, namely for the plain sequential version of RRT, which led to the execution of only 3 runs for each parametrization.

The test of subsection 5.1.2 shown that with less than 4 iterations, IPS-RRT was not able to search far enough to detect incoming corners (at least for “CG Speedway number 1”). Less than 800 states were not generated either as applying such number of states revealed to produce very inconsistent behaviour. Oppositely, a maximum of 6400 states were generated because the GPU’s resources were nearly exhausted for that number of states.

In the end, the used parameterization strategy for these tests was executing 4 iteration IPS-RRT searches alongside sequential RRT searches while doubling the number of states starting from 800 all the way up to 6400: \{800,1600,3200,6400\}.

The bot was firstly executed for 10 runs of 10000 ticks on the track “CG Speedway number 1”. The recorded search times can be consulted in the chart of Fig. 6.10. The chart reveals that although the parallel implementation scaled better than the sequential one, the overall gains of the GPU-based parallelization were low when compared with the CPU implementation. Moreover, the big standard deviation lowers this track’s test significance.

![Mean Search Times on "CG Speedway number 1"](image)

**Figure 6.10:** Chart displaying the mean search times for the track “CG Speedway number 1”.

The bot was also executed for 3 runs of 10000 ticks on the track “E-Track 5”. The recorded search times are displayed in the chart of Fig. 6.11. Once again it is remembered that all IPS-RRT searches

47
were executed with 4 iterations. These results present the same behaviour than the ones of the previous test. However, the low standard deviations give significance to the test. In this case, there is a slight improvement of the 6400 states search time when IPS-RRT is applied to the GPU. Moreover, a significant increase in the times of the special CPU implementation of IPS-RRT (referred in subsection 4.3.1) can also be seen for the same number of states. Maybe the amount of information that had to be copied to/from partial trees started to overwhelm the CPU’s cache.

![Figure 6.11: Chart displaying the mean search times for the track “E-Track 5”](image)

The bot was finally executed for 10 runs of 10000 ticks on the track “Michigan Speedway”. The recorded search times can be consulted in the chart of Fig. 6.12. The results observed for this track present once again the same characteristics as the ones before. However, the search times of both the sequential RRT and the GPU implementation of IPS-RRT were statistically the same for 6400 states. Maybe this occurred because the track has fewer segments to test in constraint checking. Therefore, the processed data was not enough to overwhelm the CPU’s cache.

![Figure 6.12: Chart displaying the mean search times for the track “Michigan Speedway”](image)
6.4 Studying the Quality of Returned Solutions

Several tests were conducted in order to study the impact IPS-RRT had on the quality of the returned solutions. Such tests consisted on acquiring the minimum and mean lap times while executing the bot in the same circumstances as in the tests of subsection 6.3.2. Because the car started while standing still, the first lap did not count for the lap times evaluation, as the first lap times are higher and therefore could influence the recovered data. As the lap time decreases, it is assumed that the bot performs better.

The mean lap times on the test track “CG Speedway number 1” while executing the bot for 10 runs of 10000 ticks can be checked in the chart of Fig. 6.14. It can be seen that there was a 19% mean lap time reduction across all “CG Speedway number 1” searches when IPS-RRT was used. This is due to the way IPS-RRT trees are built. They are expanded more in breadth than in depth when a number of iterations like 4 is applied. This leads the search to choose neighbours with better costs (bigger costs are better in this context). To help understand what causes these gains, the differences in trees layouts and path costs produced by RRT and IPS-RRT are depicted on Fig. 6.13. This figure was actually produced in runtime. The generated search states’ positions and segments’ bounds were printed to an auxiliary window.

The mean lap times on the test track “E-Track 5” while executing the bot for 3 runs of 10000 ticks can be checked in the chart of Fig. 6.15. In this case, there was a 22% mean lap time reduction when IPS-RRT was implied. Maybe the bigger gain here is due to the fact that the parametrization was pushing the car so far that minor mistakes contributed to the inflation of the sequential RRT’s lap times. Moreover the IPS-RRT lap times decreased as bigger number of states were computed. Such improvement can be justified by this track being wider and therefore having a bigger state space to explore.

The mean lap times on the test track “Michigan Speedway” while executing the bot for 10 runs of 10000 ticks can be checked in the chart of Fig. 6.16. In this case, there was just a 15% mean lap time reduction when IPS-RRT was implied. This can be explained by the fact that this track is simple to drive and therefore it is hard to commit big mistakes. Moreover, the track is very small, which makes it difficult to reduce lap times. Nevertheless, the lap times decreased as a bigger number of states were computed, just like in “E-Track 5”. Maybe this is also due to the fact that “Michigan Speedway” is wider.
than “CG Speedway number 1”.

As mentioned in the beginning of this section, along with the mean lap times, the minimum lap times were also obtained. The full test is presented at the appendix of this document, namely in part C (starting at page 63).

6.5 Discussion

The first tests revealed that executing both the AI and graphics on the same GPU was as efficient as executing the two aspects in separate GPUs. Because of that, in these circumstances, it can be said that there is a positive interaction between graphics rendering and the search algorithm processing. However, with higher graphical demanding (more recent) games, graphical rendering can be negatively influenced, especially when the GPU’s resources become nearly exhausted. This fact is very important to take in account as the amount of available search resources has to always be balanced with the available rendering resources. Nevertheless, as GPUs have been evolving at a fast pace (enforcing
the message passed on chapter 1), more and more resources become available and therefore the application of approaches such as the ones discussed here is easier.

The GPU parallel IPS-RRT scales better than the sequential RRT, because of the influence of parallelism. The CPU implementation of IPS-RRT seemed to increase its search times very fast because of the copies it makes from/to partial trees. The search times of the GPU implementation are considerably higher than the sequential versions’ times when the number of states is low. This fact emerges because GPU’s memory accesses are time consuming (as discussed on section 5.2), and because the presented time spans are very small. This algorithm implies many accesses to the GPU’s memories and that reveals to be problematic. It can be deducted that the factor that most negatively influenced the search times was precisely the discrepancy in latencies of the memories accessed by the CPU and GPU.

It is also interesting to notice how the search times become more similar as a higher number of states is used. There is however a slight deviation from the noticeable threshold ($\approx 0.0417$ of a second per frame) among all GPU searches that go beyond 800 states and CPU searches that process more than 3200 states. Despite this fact, as the number of states increased, the GPU searches deviated less from this threshold than the CPU ones.

Furthermore, as can be seen in the search times charts, the time spent on each search depended majorly on the characteristics of the tracks. Interestingly for “CG Speedway number 1”, some of the search times of the CPU implementation of IPS-RRT appear to be lower than the sequential RRT search times. This is maybe due to the fact that more states are pruned on IPS-RRT, which helps to decrease the nearest neighbour and constraint checking times. Moreover, searches on tracks with a higher number of segments were generally slower than searches on tracks with a lower number of segments.

With 6400 states, GPU IPS-RRT revealed to be more computationally efficient than the sequential RRT, although the search times produced when that happened deviated from the perceivable threshold ($\approx 0.0417$ seconds). After analysing the search times on “CG Speedway number 1” it was observed that IPS-RRT had more empty plans (implying no writes to the global memory while searching), which could be responsible for the increased computational efficiency at this track. Nevertheless, in “E-track 5” and “Michigan Speedway”, the search times were not influenced by empty searches. An example of the phenomena seen on “GC Speedway number 1” is depicted on Listing 6.1.
Listing 6.1: Phenomena seen while executing the bot on “GC Speedway number 1”. The searches that
returned empty plans (the information painted red) were much faster than the others.

<table>
<thead>
<tr>
<th>searchID</th>
<th>numStates</th>
<th>numIterations</th>
<th>run</th>
<th>searchTimes</th>
<th>lapTimes</th>
</tr>
</thead>
<tbody>
<tr>
<td>ParallelRRT:800;4;7;0.030000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ParallelRRT:800;4;7;0.031000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ParallelRRT:800;4;7;0.030000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( ... )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ParallelRRT:800;4;7;0.009000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ParallelRRT:800;4;7;0.007000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ParallelRRT:800;4;7;0.007000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( ... )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ParallelRRT:800;4;7;0.028000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ParallelRRT:800;4;7;0.030000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ParallelRRT:800;4;7;0.034000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( ... )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ParallelRRT:800;4;7;0.028000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ParallelRRT:800;4;7;0.030000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ParallelRRT:800;4;7;0.033000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ParallelRRT:800;4;7;0.034000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ParallelRRT:800;4;7;0.038000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ParallelRRT:800;4;7;0.042000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ParallelRRT:800;4;7;0.036000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ParallelRRT:800;4;7;0.041000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The results presented in this chapter also lead to believe that with more currently available GPU
resources (such as a top of the line GPU), IPS-RRT could benefit more by the use of the GPU im-
plementation, as bigger problem sizes could be tested. However, like what can be seen in the tests
of 6.3.1.2 and 6.3.1.1, there is an increase of parallel search times along the number of states due to
thread overhead and warp synchronization, and that can negatively influence such times.

As can be seen in the charts presented in Figs. 6.14, 6.15 and 6.16, there is a considerable lap time
reduction (around 19% across all tests) when IPS-RRT is either applied to the CPU or parallelized in
the GPU. In all cases, even the 800 state IPS-RRT searches reveal to proportionate lower lap times than
6400 state RRT searches. This can be explained by the fact that IPS-RRT trees tend to be expanded
more in breadth than in depth, having more children per state, which makes the algorithm pick better
neighbours for each state. This proves that by using a GPU-based implementation of RRT in racing
contexts such as this one, a significant increase in the quality of returned solutions can be achieved
even while generating a much lower number of states.

Moreover, the lap times seemed to attenuate as more states were added. Those attenuations are
possibly due to the fact that the time spent while searching reduced the forward model precision. In fact,
one of the conclusions of the preliminary test of subsection 5.1.2 was that when using a big number of
tree samples, the search took so long that the returned solution was no longer consistent with the
car’s current position (Fig. 5.3 b ). Nevertheless, while in “CG Speedway number 1” there was no
major impact of using more states, in the other tracks there is a visible improvement of lap times when
the number of generated states is increased, which (as presented before) can be due to the fact that “E-track 5” and “Michigan Speedway” have a wider drivable area (wider track) and therefore a vaster state space to be explored, namely if a closer range is considered.

By using the Visual Studio Performance Wizard with CPU sampling\(^2\) on each sequential method call of the final version it was confirmed that the time is spent mostly on constraint checking (12.55% of the samples - decrease of 1.08% compared with the first implementation presented in subsection 5.1.2). The random state generation and nearest neighbour checking procedures are much faster (they both took only a sum of 2.02% of the samples - increase of 1.81% compared to the first implementation). It can then be concluded that constraint checking continues to be the procedure that mostly affected the duration of each IPS-RRT iteration. The differences in the samples percentages between preliminary and the final versions can be explained by the changes that were applied to the constraint checking procedure. As the sequential version nearest neighbour procedure was not changed, it seems natural that such method sampling percentage increased while the other decreased. This fact still accentuates the conclusions mentioned in [2]: constraint checking seems to be the most computational demanding procedure on RRT and its variants.

Conclusions

The goal of this work is to explore and study the use of GPGPU-based parallelization techniques to improve the application of RRT to racing videogames. The work covered the implementation of a bot for the TORCS racing videogame. Although never used (as far as the author knows) as a base for the construction of TORCS bots, RRT was chosen as it is appropriate for problems which generate big search spaces and involve physics attributes like position and velocity.

Although not initially stated as part of this thesis’ goal, one important subproduct of the work was the development of a new variant of the RRT algorithm named Iterative Parallel Sampling RRT, which creates better solutions for this domain by running the main RRT loop in parallel, performing parallel sampling. As mentioned in chapter 1, this algorithm’s proposal led to the acceptance of one article in the EPIA\footnote{https://web.fe.up.pt/~epia2017/ (as consulted on Tuesday 12th September, 2017).} conference. Such article is included at the appendix of this document (it starts at page 69).

After studying the results of the preliminary tests of section 5.1 and the tests presented on chapter 6, important aspects about this particular approach were learned. They allow to answer the questions presented in section 1.1:

- Performance tests showed that executing both the AI and graphics on the same GPU for this particular domain was as efficient as executing the two aspects in separate GPUs. However, as discussed in section 6.5, with higher graphical demanding (more recent) games, the graphics rendering can be negatively influenced, especially when the GPU’s resources become nearly exhausted. Nevertheless, as GPUs are evolving at a fast pace, more and more resources become available. This eases the application of approaches such as the one discussed in this work.

  It was also observed that the search times produced when a high number of states were generated by IPS-RRT deviated from the perceivable threshold ($\approx 0.0417$ seconds). This leads to the conclusion that it is still hard, even for low specification graphical applications such as TORCS, to maintain a smooth frame refresh rate while using this kind of approach. However, as mentioned in 6.3.1.2, all the searches were fully calculated in a single frame. If the search processing was distributed between several frames instead of a single one, the search times per frame could be lowered significantly;

- In terms of computational efficiency, oppositely to what we expected, GPU IPS-RRT did not fare better than the sequential RRT or the special CPU implementation of IPS-RRT (referred in subsection 4.3.1), except when a high number of states was generated. Such comparison was concluded to be unfair, as the GPU’s memory access latencies are high everytime the off chip memories are accessed. Although such latencies can be reduced by using per block shared memories or constant memory, this work proves that it is still hard to effectively apply them. They are very limited in terms of size and therefore could not be used while executing all of the GPU IPS-RRT procedures. Moreover, such memories are not persistent and could not save data between GPU IPS-RRT iterations.
On the positive side, tests revealed that (1) while increasing the number of parallel iterations, the search times grow sub-linearly, due to the synchronization/ thread creation overheads and (2) while increasing the number of states, the search times also grow sub-linearly. In short, GPU IPS-RRT starts off worse than the CPU versions, but scales better as the number of states and samples increase. Given these results, we believe that the GPGPU-based parallelization can be more efficient when applied to problems that require a large number of samples. Consequently, it will be more efficient as the memory of graphics cards increase too;

- Finally (and more importantly), IPS-RRT achieved a noticeable increase on the quality of returned solutions (overall reduction of around 19% in lap times), even while processing a much lower number of states than the original RRT implementation. This proves that GPGPU-based techniques such as GPU IPS-RRT can be applied to improve the quality of RRT solutions in racing contexts such as the one of TORCS, without having a negative impact on the computational efficiency. In this case, IPS-RRT expanded the trees more in breadth than in depth, having more children per state and thus achieving faster plans by picking better neighbours.

7.1 Notes for Future Work

One important future work would be to apply IPS-RRT (either on CPU or GPU) to other domains or types of problem, checking where plan quality improvements similar to the ones seen in this work can be achieved.

Another important aspect to study would be to improve the nearest neighbour calculation, for instance, by partitioning the state space or using a sampling method instead of exhaustively iterating through all states. We believe that these type of techniques would allow us to mitigate the memory-caused problems of IPS-RRT. In what respects to constraint checking, one aspect to consider would be to prune states that are similar to the ones generated before, like in BSD-RRT (presented in Algorithm 4). Other improvements to IPS-RRT may include (1) the development of a mechanism to use previously computed trees instead of starting from an empty tree (allowing to compute searches along various frames) or (2) testing a model to automatically adjust the number of iterations and states to the available system resources or track characteristics.

Other studies of this type may include (1) exploring other variants of parallel RRT or even other search algorithms which can reveal more efficiency in low time spans when applying the GPGPU paradigm or (2) testing this algorithm with top of the line GPUs (as referred in section 6.5).

In what respects to the application of RRT to TORCS, some aspects can be explored such as the implementation of a competition oriented forward model which takes in account advanced racing aspects such as (1) the gears to de-accelerate the car or (2) regression analysis as in [8]. This would allow the bot to drive fast in more demanding tracks (with many accentuated bends, for example).

Future work can also approach the analysis of the behaviour of other parallel RSA and/or other type of search algorithms in graphical demanding applications. Videogames other than TORCS can be used as cases to study, remembering that this study can be extended to categories of videogames other than racing or even to other types of time constrained applications.

\[\text{Moreover, improvements were achieved either while IPS-RRT was parallelized on the GPU or executed on the CPU (through the special implementation referred in subsection 4.3.1).}\]


Appendix: Work Schedule

This work included the bot development (practical component) alongside the writing of the thesis.

A.1 Initial Work Schedule

The plan initially scheduled for the bot's development throughout the year was:

1. Build a bot using a sequential search implementation;
2. **Milestone: Test bot**;
3. Build a bot using GPGPU parallel search implementation;
4. **Milestone: Test parallel against sequential bot**;
5. Improve the bot to compete with opponents;
6. **Milestone: Test bot behavior**;

A.2 Work Log

Although a complete version of the bot that was developed alongside the writing of this document, such bot was not optimized when competing with opponents. However, like all TORCS bot implementations, it can actually be put side by side with any other bot.

The bot's development involved several stages. They include:

- 11 of November: bot development started;
- 12 of November: sequential RRT skeleton implemented;
- 12 of November - 15 of November: Began to build the initial forward model;
- 16 of November - 5 of December: Fixed some bugs;
- 6 of December - 15 of December: PID class implemented;
- 16 of December - 2 of February: Fixed some bugs;
- **(Milestone) 2 of February: First complete RRT sequential version finished**;
- 3 of February - 27 of February: Parallel version development;
- **(Milestone) 28 of February: First parallel version implemented**;
- 29 of February - 7 of March: Added log writing feature;
- 8 of March: Passed track segment linked list to an array;
- 8 of March - 11 of March: Fixed some bugs;
- 11 of March - 17 of March: Fixed controls and constraint checking bug;
- 17 of March - 24 of March: Major code restructuring. Started physical model implementation;
- 25 of March - 29 of March: Physical model implemented;
- **(Milestone) 31 of March: Complete version for EPIA paper**;
- 31 of March - 19 of April: Map tweaks for debugging;
- 20 of April: Added 2D plane projection to the graph (for the EPIA paper);
- 18 of May - 21 of May: Changed some code comments to ease understanding of the code;
- 22 of May - 23 of May: Changed some TORCS compiler settings and ran some tests;
- 24 of May - 10 of June: Overall code optimization;
- 18 of June - 21 of June: Execution of final tests.
Appendix: Full Test of Section 6.2

Chart comparing the mean search times of GPU searches ran alongside graphics computations for 800 state searches.

Chart comparing the mean search times of GPU searches ran alongside graphics computations for 1600 state searches.
Chart comparing the mean search times of GPU searches ran alongside graphics computations for 3200 state searches.

Chart comparing the mean search times of GPU searches ran alongside graphics computations for 6400 state searches.
Appendix: Full Test of Section 6.4

Chart displaying the mean lap times for the track “CG Speedway number 1”.

Chart displaying the minimum lap times for the track “CG Speedway number 1”.

Chart displaying the mean lap times for the track “E-Track 5”.

Chart displaying the minimum lap times for the track “E-Track 5”.

Chart displaying the mean lap times for the track “Michigan Speedway”.

Chart displaying the mean lap times for the track “Michigan Speedway”.

64
Chart displaying the minimum lap times for the track “Michigan Speedway”.
Appendix: Code Snippets

D.1 RRT Interface (RRT.cuh)

Note: Although being called RRT, it can be applied to any search. This name was used only to serve its purpose in this work’s context.

```cpp
#pragma once
#ifndef RRTSTAR_H
#define RRTSTAR_H

#include "State.cuh"
#include <vector>

// ----------------- RRT Base class ---------------------------
// This class is an general implementation of the planning
// module. It is the base class for SeqRRT and ParRRT.
// -----------------------------------------------------------

class RRT
{
public:
    RRT() {} 
    virtual ~RRT() {} 
    virtual std::vector<State*> search() = 0;
    virtual std::vector<State> getGraph() = 0; // for debug purposes

    virtual char* getSearchName() = 0; // for debug purposes
};
#endif;
```

D.2 GPGPU IPS-RRT class header (ParRRT.h)

```cpp
#pragma once
#ifndef PARRRTSTAR_H
#define PARRRTSTAR_H

#include "Kernel.cuh"
#include "RRT.cuh"
#include "Heuristics.cuh"
#include <robottools.h>
#include <iostream>
#include <algorithm> // std::transform
#include <time.h>
```
// ----------------- ParRRT Search class ---------------------------
// This class represents the GPGPU parallel search procedure which
// implements the RRT base class virtual methods. This class
// calls the CUDA kernel in the search() method
//----------------------------------------------------------------------
class ParRRT : public RRT {
private :
    // vars
    // initialization vars
    tPolarVel maxCarAcceleration;
    double actionSimDeltaTime;
    int nTrackSegs;
    int nIterations;
    State* initialState;
    int forwardSegments;
    State* graph; // the returned search tree!

    // kernel vars
    State* kernelGraph;
    tPolarVel* kernelVelArray;
    tSimpleTrackSeg* kernelSegArray;
    int numKernelBlocks;
    int numKernelThreadsPerBlock;

public :
    // methods
    ParRRT(State initialState, int nIterations, State* kernelGraph, tPolarVel*
            kernelVelArray, tSimpleTrackSeg* kernelSegArray, int nTrackSegs, double
            actionSimDeltaTime, tPolarVel maxCarAcceleration, int numKernelBlocks,
            int numKernelThreadsPerBlock);
    ~ParRRT();
    std::vector<State*> search();
    std::vector<State> getGraph();
    char* getSearchName();
};

D.3 Sequential RRT class header (SeqRRT.h)

#pragma once
#ifndef SEQRRTSTAR_H
#define SEQRRTSTAR_H
#include "RRT.cuh"
#include "Heuristics.cuh"
#include <robottools.h>
#include <iostream>
#include <time.h>

// ------------------ SeqRRT Search class --------------------------
This class represents the sequential search procedure which implements the RRT base class virtual methods.

```cpp
class SeqRRT : public RRT{
private: //vars
    //initialization vars
    tPolarVel maxCarAcceleration;
    double actionSimDeltaTime;
    int startSegIndex;
    int finalIndex;
    tSimpleTrackSeg* trackSegArray;
    int nTrackSegs;
    int nIterations;
    State initialState;

    //aux vars
    double maxCost;
    State bestState;

    State* graph; // the returned search tree!
    unsigned int graphSize;
    int graphIterator;

private: //methods

    //graph operations
    void initGraph();
    void resizeGraph(unsigned int newSize);
    void deleteGraph();
    void pushBackToGraph(State &element);

    //main loop procedure
    void generateStates(double nIterations);
    State generateRRT();

public: //methods
    SeqRRT(State initialState, int nIterations, tSimpleTrackSeg* trackSegArray,
            int nTrackSegs, double actionSimDeltaTime, tPolarVel maxCarAcceleration);
    ~SeqRRT();

    std::vector<State*> search();
    std::vector<State> getGraph(); //for debug purposes
    char* getSearchName();
};
```
Abstract. Graphics Processing Units have evolved at a large pace, maintaining a processing power orders of magnitude higher than Central Processing Units. As a result, the interest of using the General-Purpose computing on Graphics Processing Units paradigm has grown. Nowadays, big effort is put to study probabilistic search algorithms like the Randomized Search Algorithms family, which have good time complexity, and thus can be adapted to massive search spaces. One of those algorithms is Rapidly Exploring Random Tree (RRT) which reveals good results when applied to high dimensional dynamical search spaces. This paper proposes a new variant of the RRT algorithm called Iterative Parallel Sampling RRT which explores the use of parallel computation in GPU to generate faster solutions. The algorithm was used to construct a CUDA accelerated bot for the TORCS open source racing game and tested against the plain RRT. Preliminary tests show lap time reductions of around 17% and the potential for reducing search times.

Keywords: General-Purpose computing on Graphics Processing Units, Randomized Search Algorithms, Rapidly-exploring Random Trees, The Open Racing Car Simulator, Planning

1 Introduction

The need to search big continuous state spaces led to the creation of stochastic search algorithms (algorithms that have a probabilistic approach on search) like the Randomized Search Algorithms (RSA) family. Most known RSA include Rapidly-Exploring Random Tree (RRT) [13], R* [7] (based on A*) or Monte-Carlo Tree Search (MCTS) [4]. In particular, RRT can adapt to rapidly changing worlds and/or high dimensional state spaces, dynamically generating the representation of those spaces. Besides that, it has a good behaviour for non holonomic problems with non holonomic constraints. These are characterized by implying variables that change with time (not time itself), like gravity or angular velocity. 2 Problems which are defined by manipulating force attributes like velocities or accelerations.

---

1 Problems with non holonomic constraints. These are characterized by implying variables that change with time (not time itself), like gravity or angular velocity.

2 Problems which are defined by manipulating force attributes like velocities or accelerations.
To speed up the exploration of massive state spaces, major effort has recently been put to use the General-Purpose computing on Graphics Processing Units (GPGPU) programming paradigm to create parallel versions of RSA, namely MCTS ([2],[11]), R* [6] and even the RRT itself ([1], [5]). However, those parallelizations have not been much explored in real time applications such as videogames, where the processing time assigned to the search algorithm needs to be balanced with the execution of high graphical requirements. In order to achieve good results on time constrained applications such as those, a parallel version of RRT using the GPGPU paradigm is proposed, the Iterative Parallel Sampling RRT (IPS-RRT).

To analyse and test the properties of IPS-RRT we implemented it in CUDA and applied it to a car racing videogame extensively used for academic research, The Open Racing Car Simulator (TORCS)\(^3\), which is a multi-platform open source game that serves as a base for the construction of AI controllers. Racing games can benefit from the use of RRT versions like IPS-RRT, as directional speed and steering generate a big continuous state space.

## 2 Related Work

Several techniques have been applied to improve the performance of RRT. RRT*, referred in [1], is an optimization of the RRT algorithm to asymptotically converge to a better solution. It achieves this through multiple neighbour selection and node rearrangement. A big drawback of RRT* is that it implies a big number of dependencies as the additional phases use previously calculated information. This makes it much less prune to parallelization than the plain RRT. Constraint parallelization [1] also helped to improve RRT’s performance by parallelizing the process that checks constraints and culls invalid points. When searching potential object collisions in the motion planning problem, parallel threads checked the collision with each obstacle concurrently. This approach has the drawback of only parallelizing one part of RRT’s execution. If only one obstacle is considered there is no gain because the only thread is going to process the collision of all RRT states. Also, a parallelization method called Sampling-based Roadmap of Trees (SRT)[10] parallelizes RRT by creating several independent trees in parallel, joining them to produce a global tree. That tree can then be searched to find a solution. Another approach for parallelizing RRT is presented in [5], Bulk Synchronous Distributed RRT. In that algorithm, processes create several RRT samples and synchronize them to a global tree through message broadcasting. Approaches such as the ones in [5] revealed to be applied in distributed computation contexts.

Although, to our knowledge, no work applied RRT to TORCS, other approaches have been explored. An Inverse Reinforcement Learning bot [3] was developed with the objective of testing a framework for autonomous road driving. This approach involved two sub controllers: a speed and pose sub controllers.

\(^3\) [http://torcs.sourceforge.net/](http://torcs.sourceforge.net/) (as consulted on June 20, 2017)
An MCTS racing bot [4] was also developed, that uses a state space based on physical aspects like positions and velocities. The bot uses that spatial representation to predict future states through a physics based forward model.

### 3 Iterative Parallel Sampling RRT

Instead of applying constraint parallelization (like in [1]) or creating parallel trees (like in SRT [10]), IPS-RRT focuses on executing several iterations of parallel tree samples. At each iteration, a number of samples are concurrently generated and checked for constraints (using a thread for each sample). The valid samples are added to local trees which are then synchronized to the global tree at the end of each iteration. IPS-RRT’s pseudocode is presented in Algorithm 1.

**Algorithm 1 IPS-RRT**

1: procedure GenerateRRT(numIterations,numParallelSamples,T)
2:     loop from 0 to numIterations:
3:         launch numParallelSamples threads computing this:
4:             $T' \leftarrow T$
5:             $x_{rand} \leftarrow \text{randomState}()$
6:             $x_{near} \leftarrow \text{nearestNeighbor}(x_{rand},T')$
7:             $x_{new} \leftarrow \text{applyDelta}(x_{rand},x_{near})$
8:             if not validPoint($x_{new}$) then
9:                 endThread()
10:             add vertex $x_{new}$ to $T'$
11:             add edge ($x_{near},x_{new}$) to $T'$
12:     synchronizeThreads() , $T \leftarrow$ all $T'$s
13: return $T$

IPS-RRT receives three parameters: (1) the number of iterations, (2) the number of concurrent samples to be generated in each of those iterations and (3) an empty search tree $T$ that corresponds to the global tree. Partial trees (represented by $T'$) are used inside each thread. They consist in snapshots of the contents of $T$ at the start of the iteration (line 4).

A state $x_{rand}$ is randomly generated (line 5). Then, the algorithm searches for the closest state $x_{near}$ to $x_{rand}$ in the tree $T'$ (line 6). However, $x_{near}$ is not added directly to the tree. In order to expand the tree in a controlled way, a new state $x_{new}$ is generated by moving $x_{near}$ a small distance $\delta$ in the direction of $x_{rand}$ (line 7). If the new state is not valid (does not verify the problem constraints), it is not added to $T'$ (nor to $T$) and the thread stops its execution (lines 8 and 9). Otherwise, the new state $x_{new}$ is added to $T'$ (line 10) along with a new edge between $x_{near}$ and $x_{new}$ (line 11) to represent a parent/child connection between the two states. At the end of each iteration, threads are
synchronized and the samples generated in each thread are copied over to $T$ (line 12). Lastly, the algorithm returns the global tree $T$ (line 13).

It is important to compare our proposed algorithm with Bulk Synchronous Distributed RRT (BSD) [5] because it also parallelizes the build of a single RRT tree. However, the process of creating the trees is different, because of the way IPS-RRT’s schedule works. IPS-RRT iteratively builds partial trees by thread synchronization instead of incrementally broadcasting them in a distributed process context. IPS-RRT process is more adequate for a GPGPU implementation, as message passing would not efficiently work in such contexts. Another difference is that in BSD broadcast is done only after a number of new valid states are added, while in IPS-RRT synchronization occurs after each thread processes a sample independently of being added to the tree or not. The reason for this change is that since the synchronization must wait for the slowest thread, it would not be appropriate to wait for an unlucky thread processing several invalid samples.

3.1 IPS-RRT Characteristics

By selecting different number of iterations and number of parallel samples, IPS-RRT presents different characteristics. Fig. 1 illustrates how IPS-RRT trees can significantly differ from the classic RRT trees.

If one tries to maximize the parallelization by having just one iteration and performing all samples in parallel, then all resulting new states will connect to the initial state (given that each partial tree only has the initial state). This is illustrated in the second leftmost picture from Fig. 1. By increasing the number of iterations, the algorithm will increase the span of the search tree, and will be able to reach more distant states. When the number of iterations is equal to the number of RRT generated samples and only one sample is created for each iteration, then the IPS-RRT becomes equivalent to the traditional RRT, as illustrated in the rightmost picture from Fig. 1.

4 TORCS Bot Implementation

This section presents the implementation of our developed CUDA accelerated TORCS bot using IPS-RRT. The solution architecture is presented and its components analysed. The state representation and restrictions are detailed and depicted for better reader comprehension.

4.1 Bot Architecture

Our TORCS bot consists of several modules, similar to the approach in [12]. A schematic representation of the solution can be found in Fig. 2. It includes:

---

4 By thread synchronization it is meant that the threads wait for the slowest one before the execution continues, just like a barrier.
A planning module which is composed by IPS-RRT as well as the plain RRT (for comparison purposes);

A control module which receives the plan from the planning module and coordinates the actions needed to drive the car by calling TORCS back-end;

The TORCS back-end which provides the necessary game core procedures to describe and control the car and query track information.

The planning module is called when the control module checks that either the car passed the last point of the current intermediate plan or a time limit is reached. When the planning module returns, the control module follows the new intermediate plan. This procedure is repeated until the end of the race.
**Planning Module** The planning module consists of an abstraction for the execution of both the plain RRT and IPS-RRT. The output of this layer is the best plan found without the initial point (the initial point represents the current state which is redundant to seek).

**Control Module** The module that controls the car is divided in two components: the pedals component and the steering component (similarly to the speed and pose controllers described in [9]). It drives the car using the TORCS control parameters depicted in Fig. 4.

The pedals are controlled by a Proportional–Integral–Derivative (PID) controller\(^5\) as the relation between their position and the car’s acceleration varies from car to car and cannot be directly acquired. The objective of using PID is to minimize the errors incurred by the search, getting practical action results close to the theoretical predictions. The use of this type of controller revealed to be a good compromise. There was no need for a steering PID controller given that the steering angle can be directly determined as described in Fig. 3.

![Fig. 3. Used method to determine the steering angle. The target search point is represented in blue. The position of the car is displayed in black along with the car direction vector. The angle between the car’s direction and the current state position, displayed as $\theta$, is obtained by the arc-tangent of the legs of the triangle.](image)

4.2 TORCS Search State Representation

The objective of each state is to represent a predicted car situation. Thus, the representation that seemed more adequate was including attributes like position and velocity. While the position of the car is represented in the cartesian space defined by the TORCS back-end, the velocity is represented in a polar space aligned to the same cartesian space. The function of each state attribute is clarified by the scheme provided in Fig. 4.

\(^5\) The article [8] introduces PID and refers how to generally implement and tweak a PID controller so that it can adapt to a racing environment
4.3 IPS-RRT Applied to TORCS

This subsection details the implementation of the IPS-RRT methods presented on page 3. First, a velocity is randomly sampled and used to generate a new state $x_{rand}$ (line 5 of IPS-RRT’s pseudocode). A forward simulation method considers a simple physical model to generate the position of $x_{rand}$ based on its velocity. The position is obtained by multiplying the velocity by a fixed search action time. The nearest neighbour $x_{near}$ (line 6) is determined by choosing the state which has the lowest velocity angle variation to $x_{rand}$ 6. To calculate $x_{new}$ (line 7), a set of intermediate states are generated between $x_{near}$ and $x_{rand}$. Such states are computed by interpolating their velocities. Fig. 5 illustrates the procedure. Finally, $x_{new}$’s validity (lines 8 and 9) is checked by verifying if all intermediate states are positioned inside the track. After the search tree is built and returned (line 13), the best tree state is picked as the one estimated to maximize the covered track distance 7 in the minimum amount of time. This is calculated by the ratio between the track distance and the depth of the state in the tree 8. The best state is then backtracked to the initial state and the corresponding path is returned as the solution.

---

6 The angle variation between two velocity vectors is the minimal angle between them.  
7 The track distance for a state $s$ is determined by projecting the line segment between the initial state and $s$ along the centre of the track, and calculating the length of the projection.  
8 Since a connection in the tree represents a fixed time span, the depth of the state in the tree can be used as an estimation of the time required to reach that state.
5 Evaluation

In this section, the hardware and software used to develop and test this work are presented, along with the analysis of several preliminary tests results. These tests were performed using preliminary versions of the algorithms which were not yet optimized. The goal was to determine if there is a potential advantage when using the explored approach. A screenshot of the bot running the test cases can be seen in Fig. 6.

5.1 Tools

The work was developed in the C++ programming language, using Microsoft Visual Studio\textsuperscript{9} 2013 and CUDA development toolkit 7.5 \textsuperscript{10}. This combination

\begin{itemize}
\item \textsuperscript{9} https://www.visualstudio.com/ (as consulted on June 20, 2017)
\item \textsuperscript{10} https://developer.nvidia.com/cuda-toolkit (as consulted on June 20, 2017)
\end{itemize}
was chosen because despite more recent ones being available, it was the most stable and well documented when the development started.

The used graphical card was a Nvidia GTX 960M (laptop) GPU with 4GB of VRam (which is a middle range laptop gaming GPU). It has a 5.0 compute capability and a 1.176 Ghz clock rate. The used CPU was an Intel(R) Core(TM) i7-4720HQ (laptop) CPU with a 2.6GHz clock.

5.2 Performance of one search procedure

The mean of search times (around 3 laps) for both implementations on the representative road track “Speedway-1” can be consulted in the chart of Fig. 7.

![Chart displaying the mean search times.](image)

IPS-RRT appears to be more time consistent along lap searches than the sequential version, because of the way the parallel schedule works. The overall search time seems to be reduced in practically all cases, supporting the idea that the parallel implementation can improve the efficiency. There are however situations where the GPU IPS-RRT implementation is actually slower than the sequential RRT search, so more tests are required in order to properly understand in which situations is preferable to use the GPU implementation of IPS-RRT.

By using the Visual Studio Performance Wizard with CPU sampling[^11] on each sequential method call in the whole TORCS execution, it was confirmed that the time is spent mostly on constraint checking (13.63% of the samples). The random state generation and nearest neighbour checking procedures are much faster (they both took only a sum of 0.21% of the samples). It can then be concluded that constraint checking is the procedure that mostly affected the duration of each IPS-RRT iteration.

5.3 Bot Racing Performance

This metric is tested by examining the lap times produced by the bot. As the lap time decreases, it is assumed that the bot performs better. A mean of three lap times on the test track “Speedway-1” regarding various searches can be consulted in the chart of Fig. 8. The acronym “NC” (Not Consistent) means that the search executed some recoverable mistakes (the data was measured on three non-consecutive clean laps). A recoverable mistake means that the car gets out of track but can return by itself. The acronym “DNF” (Did Not Finish) means that the search failed to finish a lap, committing an unrecoverable mistake (like when the car gets out of track and the search span is not big enough to bring it back).

![Mean Lap Times (in seconds)](chart.png)

**Fig. 8.** Chart displaying the mean lap times.

As can be seen in the chart presented in Fig. 8, there is a considerable lap time reduction (around 17% across all tests) when IPS-RRT is applied, as the generated paths are faster. However, the bot gets more inconsistent as less iterations are applied (the extreme being only one). This happens because the search span gets so small that the algorithm is not able to look ahead and detect an incoming curve. Nevertheless, with enough iterations and parallel threads, consistency can be achieved. Although non consistent, less iteration searches produced slightly better results (whenever they could) because the smaller search time decreased the prediction model error. All in all, the most successful parametrization for this scenario was using 800 states search with four IPS-RRT iterations (of 200 parallel samples).

6 Discussion

Based on the experience gained with the work described in this paper, some conclusions can be taken about IPS-RRT’s characteristics. First of all, IPS-RRT is a simple and straightforward way of dividing the work done in the sequential RRT. Secondly, when the number of iterations is low, IPS-RRT is not applicable...
to problems where a distant solution needs to be calculated in the first search, because in that case, the generated trees are small and compact (as presented on subsection 3.1). However, in problems where the solution can be obtained by determining smaller intermediate plans and executing them (as in the case of TORCS), the previous IPS-RRT limitation is not as problematic and IPS-RRT becomes a good choice as a global solution can be obtained by a composition of solutions of smaller span plans.

Some aspects about the particular application of IPS-RRT to TORCS (using the hardware presented in subsection 5.1) were discovered. When parallelizing IPS-RRT, a low number of threads is not useful, as thread overhead overwhelmed the gains obtained by parallelization. When using a low number of IPS-RRT iterations, the algorithm was not able to search far enough to detect incoming corners (left picture of Fig. 9). Another problem arose when using a big number of tree samples. Given that TORCS simulation is frame independent, the search took so long that the returned solution was no longer consistent with the car’s current position (right picture of Fig. 9).

It is important to understand that in frame independent applications like TORCS, IPS-RRT’s benefits are tightly coupled to the used hardware. If more resources are available, best search and lap time reductions can possibly be achieved, because the problems presented in Fig. 9 emanate directly from the number of IPS-RRT samples that can be processed in a given amount of time.

![Fig. 9. Occurrences that can increase lap times.](image)

**Conclusion and Future Work**

In this paper, we presented a new Graphical Processing Unit (GPU) parallelization technique for RRT and described how we created an effective bot for the game TORCS. Our proposal, Iterative Parallel Sampling RRT, seems to create faster solutions by running the main RRT loop in parallel, and performing parallel sampling. Our preliminary results indicate that the GPU implementation of IPS-RRT can achieve an increase on racing performance (reduction of around 17% in lap times), but also shows potential for improving RRT search times.
However, a longer and more detailed evaluation of the system is required to further corroborate these findings.

Additional possible improvements to this work include optimizations to the algorithm implementations, such as improving the constraint checking and nearest neighbour processes or the development of a mechanism to use previously computed trees instead of starting from an empty tree. Another interesting idea would be to create and test a version of the algorithm that automatically adjusts the number of iterations to the available system resources.

Acknowledgments This work was supported by national funds through Fundação para a Ciência e a Tecnologia (FCT) with reference UID/CEC/50021/2013.

References