Application and Design of GPU Parallel RRT for Racing Car Simulation

Author: Samuel Simão Canada Gomes
Supervisors: Prof. João Dias, Prof. Carlos Martinho

Instituto Superior Técnico, Lisbon, Portugal
WWW home page: https://tecnico.ulisboa.pt/

Abstract. The GPU systems have recently evolved at a large pace, maintaining a processing power orders of magnitude bigger then CPU's. As a result, the interest of using the GPGPU paradigm has grown. Nowadays, big effort was put to study probabilistic search algorithms like the RSA family, which have good time complexity, thus can be adapted to big search spaces. One of those algorithms is the RRT which reveals good results when applied to high dimensional dynamical search spaces, such as in robot motion planning. The work here presented focuses on the use of a GPU parallel version of an RRT inspired algorithm to build an effective bot for the open source racing game TORCS, as there is big interest to study the application of GPU to create effective AI also delivering a smooth graphical frame actualization rate.

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

1 Introduction

1.1 Motivation

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.), contributed to the development of an architecture that can process general information, diverging from traditionally processing exclusively 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) in the November of 2006, according to [1]. As a result of this technological progress, 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

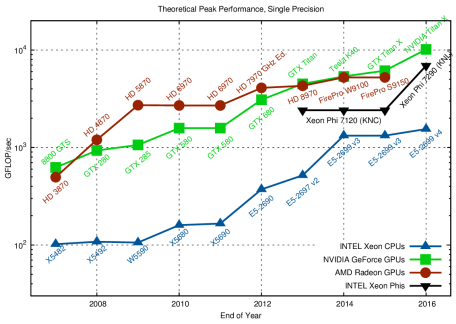

Fig. 1. Comparison between the CPU and GPU computing capabilities (in Gflops/s) through time[1].

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), like the *Randomized Search Algorithms* (RSA) family, which only consider representatives of the search space. As a result, major effort has recently been put to use the GPGPU programming paradigm to create parallel versions of RSA.

1.2 Problem

Many studies on improved versions of known search algorithms, namely RSA, focus on improving their performance by recurring to the GPGPU paradigm. Those studies are generally applied to classical AI problems like in the *Traveling Salesman Problem* (TSP) [2] or in board games like GO [3] . However, this approach has 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.

That said, there is a problem that has to be explored: *How to exploit the use of GPGPU paradigm to improve AI behavior, maintaining graphical efficiency.* By "maintaining graphical efficiency" it is meant that the same importance will be given to both the aspects above.

1.3 Hypothesis

Most known RSA include the *Rapidly-Exploring Random Tree* (RRT) algorithm, cited from [18], the R* algorithm cited from [11] (based on A*) or the well known *Monte-Carlo Tree Search* (MCTS), cited from [8]. In particular, the RRT algorithm is very interesting, as it can adapt to rapidly changing worlds and/or

[1] Source (short URL): `http://tinyurl.com/z2g29qu`
[2] `http://www.math.uwaterloo.ca/tsp/problem/`
[3] `https://boardgamegeek.com/boardgame/188/go`

high dimensional state spaces, dynamically generating the representation of those spaces. One proposed improvement of RRT is RRT*, cited from [2], which can asymptotically converge to better solutions.

Racing videogame AI can benefit from the use of parallel versions of algorithms such as the RRT family mentioned above, 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 state space generated is commonly vast, because of the large number of possible actions that can be taken in each moment (considering for example analog steering, throttling and breaking).

Taking that fact in account, the hypothesis is to *use one RRT inspired GPU parallel implementation as a base to build a better performing bot.*

As the chosen application is related to racing, it feels appropriate to define the hypothesis mention "better performing bot" as a faster bot (travels more distance at a time span). That said, the problem term "behavior" can also be defined as the bot's driven distance and speed.

A racing videogame extensively used for academic research continues to be *The Open Racing Car Simulator* (TORCS)[4], a multi-platform open source game, which serves as a base for the construction of AI controllers. A screenshot of the TORCS game can be viewed in Fig. 2. Considering this fact, this work will use the TORCS videogame as a case to study.

Fig. 2. Screenshot of TORCS gameplay.

1.4 Objectives/Expected Contributions

The objective of this work is to make a study that compares, analyzes and improves the RSA family by expanding its applications to videogames, namely racing videogames, so that it helps the scientific community consider new contexts and techniques for applying state-of-the-art technologies.

This work can be divided into several stages:

[4] `http://torcs.sourceforge.net/`

1. Survey of what has been done in terms of GPU parallelization not only for other RSA but also for other search algorithms;
   Survey of what has been done in terms of AI controllers for TORCS;
2. Definition of an implementation architecture both for the sequential and the GPGPU versions;
3. Building a bot for TORCS using the sequential and GPU parallel versions of the RRT inspired algorithm;
4. Comparison of the sequential version bot's execution with its GPU parallel version; Comparison of this approach with other state-of-the-art approaches, checking how it stands.

1.5 Outline

The remainder of the paper is organized as follows:

- Chapter 2 starting on page 4 discusses:
  1. Some aspects that help the reader understand this approach, such as the components of a GPU and how GPGPU works;
  2. What are the state-of-the-art search algorithms that already use this technology;
  3. Which algorithms were used in TORCS;
  4. Which aspects of those algorithms can be considered for this work;
- Chapter 3 starting on page 17 details:
  1. The approach, namely the representation that will be used for the search states, the parallelization methods analyzed and the global contextualization of the work;
  2. The architecture for the bot's development;
  3. The development tools that are going to be used.

2 Related Work

In this chapter, RRT is detailed along with some of its variants, as it was chosen as the base algorithm for the work here described. Some GPGPU related aspects like the physical architecture of GPU boards, the CUDA programming model or the search algorithms improved by this technique also help the reader better understand the parallelization paradigm approached here.

2.1 RRT Algorithm

The algorithm mentioned here was created in the year 1998, documented in [18]. Because of its adaptability to rapidly changing worlds and high-dimensional state spaces, it has been massively applied to robot motion planners. A massive parallel version is also referred in [2].

This algorithm has a good behavior for non holonomic problems, which are problems that have non holonomic constraints. Non holonomic constraints are

characterized by implying variables that change with time (not the time itself), like gravity or angular velocity. These special variables are commonly notated with an over dot as in $\dot{x}$.

RRT is also keen to be applied in kynodinamic problems, which are defined by manipulating force attributes like velocities or accelerations. *These problems are generally associated with constraints such as dynamic obstacle avoidance, a very important aspect of this work as it is one of the main goals on a racing bot implementation.*

To better understand each of the algorithm steps, the sequential version is described bellow along with references to its pseudo-code (represented at Algorithm 1):

- The Algorithm initializes an empty graph with the initial search state (line 2 of the pseudo-code);
- A loop is programmed so that the number of iterations is bounded and the search time is not randomized (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 graph's state closest to that random state is picked as the closest neighbor* (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 graph and an edge is created between it and the closest neighbor (lines 8 and 9 of the pseudo-code).
- While the iterations limit is not reached, loop again.
- When the limit is reached the graph is returned (line 10 of the pseudo-code).

Algorithm 1 Original RRT

```
1: procedure GenerateRRT
2:     Init Graph G with x_init
3:     loop from 1 to k:
4:         x_rand ← randomState()
5:         x_near ← nearestNeighbor(x_rand, G)
6:         u ← selectInput(x_rand, x_near)
7:         x_new ← newState(x_near, u, δt)
8:         add vertex x_new to G
9:         add edge (x_near,x_new,u) to G
10:    return G
```

RRT* Algorithm RRT* is an optimization of the RRT algorithm to asymptotically converge to a better solution. It is referred in [2]. To better present this algorithm, the pseudo-code can be seen in Algorithm 2. One difference between RRT* and the original RRT is that in RRT*, the loop checks to see if one of the neighbor states can be part of a path that costs less than the one given by direct choice of the nearest neighbor (loop from line 9 to 13). Another difference is that the neighborhood edges relative to the new node are rearranged to reduce path cost (loop from line 14 to 19).

Algorithm 2 RRT*

```
 1: procedure GenerateRRT
 2:     Init Graph G with x_init
 3:     loop from 1 to k:
 4:         x_rand ← randomState()
 5:         x_nearest ← nearestNeighbor(x_rand, G)
 6:         x_min ← x_nearest
 7:         c_min ← Cost(x_nearest) + Cost(x_nearest to x_rand)
 8:         X_near ← nearestNeighbors(x_rand, G)
 9:         loop for each x_curr in X_near:
10:             c_curr ← Cost(x_curr) + Cost(x_curr to x_rand)
11:             if (c_curr < c_min) then
12:                 x_min ← x_curr
13:                 c_min ← c_curr
14:         loop for each x_curr in X_near except x_min:
15:             c_curr ← c_min + Cost(x_rand to x_curr)
16:             if (c_curr < Cost(x_curr)) then
17:                 remove edge {x_curr, x_curr.parent} from G
18:                 x_curr.parent ← x_rand
19:                 add edge {x_rand, x_curr} to G
20:     return G
```

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 less prune to parallelization than the plain RRT.*

A GPU implementation of this algorithm can also be found in [2] where it was applied to robotic motion planning which is, as mentioned before, a classical well studied application for the RRT algorithm. In that work, the parallelization effort was put to the collision checking procedure which revealed to be the bottleneck throughout the execution of the sequential version. It was concluded that 99% of the instructions in the *iterate* method were associated with that procedure.

RRT Applied to the game *Geometry Friends* (GF) One similar sequential version of the algorithm was applied to GF[5]. This work is documented in [17].

[5] `http://gaips.inesc-id.pt/geometryfriends/`

The approach was divided in two phases, the planning and the control. The search space was represented by special points which included not only positions of the map, but also a variable representing a gameplay situation (the *type* variable). Examples of the type of a point included situations when the player had to turn (turning point) or to jump (jump point).

In the planning phase, the RRT algorithm was used to generate a random tree so that a plan could be formed by backtracking the best tree node. The number of iterations of the RRT algorithm was bounded to 1000 to avoid random execution time. When no solution was found in that number of iterations, the best plan was returned to allow the planning module to return everytime.

The *type* described the situation a character was in, so that a correct action could be executed by the control unit. *One of the control units used was a Proportional–Integral–Derivative actuator, which followed the points present in the plan in real-time.* The operation of such controllers is detailed in *Vehicle PID Controller*, part of Section 2.3 of this document.

To clarify the functioning, the controller scheme displayed in the article can be viewed in Fig. 3 and the pseudo-code relative to the RRT implementation for this approach can be observed in Algorithm 3.

Algorithm 3 GF applied RRT

```
1: procedure GENERATERRT
2:     G ← NewGraph(x_init)
3:     best ← x_init
4:     loop for i ← 1 to MAXITER do:
5:         x ← GetAvailableState(G)
6:         x_0 ← ApplyTactics(x, G, D)
7:         if Validate(x_0, x, G, D) then
8:             G.addNewNode(x_0)
9:             if BetterThan(x_0, best) then
10:                best ← x_0
11:            if IsGoal(x_0) then
12:                return TraceBack(x_0, G)
13:        return TraceBack(best, G)
```

In Algorithm 3, there are two highly important methods: *ApplyTactics* and *Validate*.

While the *ApplyTactics* method simulates new states based on an action and current state, the *Validate* method checks if the state is valid according to the map limits of the level and if the agent did not go through an obstacle to get there.

2.2 About GPGPU

Some aspects related to the construction of GPU's help to give context and enrich the reader with the necessary knowledge for the comprehension of the GPGPU

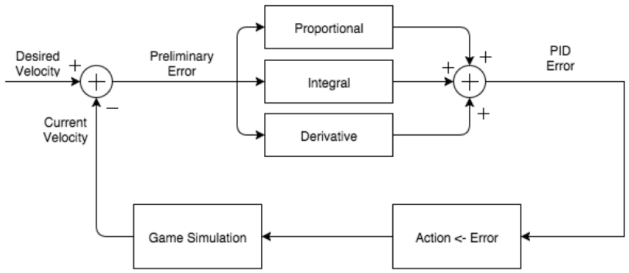

Fig. 3. Scheme of the PID controller loop used for GF. Source: [17]

paradigm, the base for the work presented. The search algorithms currently being GPU parallelized are also referred, as their description helps to identify which parallelization techniques are being explored nowadays.

GPU as a Circuit Circuit wise, a GPU is a very different chip from a CPU, because the purpose is completely different. While CPU's *Arithmetic and Logical Unit* (ALU)[6]'s are built for dealing 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. It is for that reason that the control unit on a CPU is so much bigger compared to a GPU's unit. In other words it is said that *the CPU is based on the* Multiple Instructions Multiple Data *(MIMD) model and the GPU is based on the* Single Instruction Multiple Data *(SIMD) model* [7].

To make it more clear, a diagram of both architectures is shown in Fig. 4.

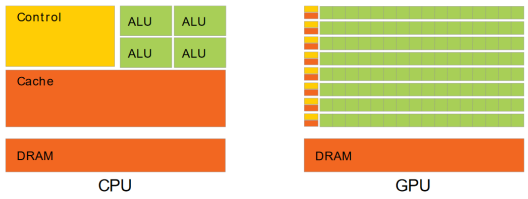

Fig. 4. Scheme of the CPU (on the left) and GPU (on the right) architectures. As can be seen, the yellow control area is much bigger on the CPU. Source:[1]

CUDA Programming Model

[6] `http://techterms.com/definition/alu`
[7] `https://www.techwalla.com/articles/differences-between-simd-and-mimd`

– **Thread Hierarchy** Unlike what happens with CPU threads, GPU threads can be coupled in two dimensional or three dimensional blocks. 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. 5;

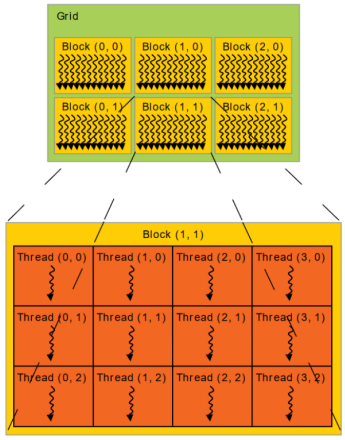

Fig. 5. Scheme of the GPU thread hierarchy. Two dimensional blocks and their threads are represented. Source:[1]

– **Memory Hierarchy** GPU's memory is divided in several modules that 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 [8] . A memory hierarchy scheme can be seen at Fig. 6;

Search Algorithms Improved by GPGPU Nowadays GPGPU is being applied to a diverse type of search algorithms, from local searches to best-first searches or even on RSA, namely the MCTS or R*. A list presenting a summary of the used parallelization procedures can be read bellow:

– **In what concerns to local search, GPGPU is applied:**
  • **On random restart hillclimbing** [14], where independent climbs are parallelized between blocks and the evaluation procedure is parallelized across threads within a block. The work optimized a 2-Opt TSP Solver (2-Opt is the name given to an heuristic that can help finding a solution

[8] Examples of their uses can be seen in [1]

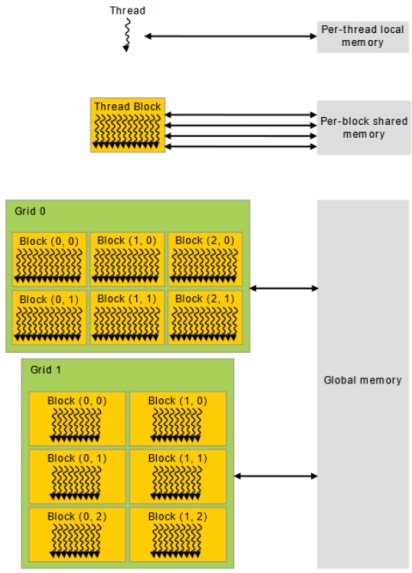

Fig. 6. Scheme of the GPU memory hierarchy. Source:[1]

for the TSP problem by rearranging the edges so that new configurations are produced);

- **On simulated annealing** [7] by decomposing the operations of the cost function, slicing the search space in several domains or by executing multiple Markov chains asynchronously. The algorithm is then tested on several function optimization problems (like the minimization of the normalized schweffel function), where it proved to be fast and efficient;
- **On genetic algorithms** ([9] and [20]) by parallelizing spacial population structure, the crossover operator and hillclimbing strategies. In the parallel implementation, each individual is active and not acted on. The implementation of the Parallel Genetic Algorithm present on [20] can be defined in two steps:
  - Each individual does local hillclimbing and selects a partner for mating in its neighborhood;
  - The offspring does local hillclimbing: it replaces the parent if it is better in some criterion (acceptance);

  This implementation was tested against other algorithms in TSP competition, where it proved to be better, namely when compared to a hybrid approach combining greedy search, simulated annealing and exhaustive search. The parallel implementation also got better results compared to other known heuristics in the Graph Partitioning Problem (a harder problem to solve then the TSP).
- **On the Alpha-beta pruning algorithm** [19]. As the implementation used *Principal Variation Splitting* (PVS), the parallelism was employed

at the nodes on the principal variation (i.e. the left-most path in the heuristically ordered game tree). A scheme that helps the reader understand PVS can be observed in Fig. 8.
In the GPU-based variant, the left-most child of each node in the principal variation was searched on the CPU to establish the lower bound for the node values. To make the parallelization possible, caution was taken to develop a version of the algorithm that was not recursive.
In this case, the game Reversi was used to test the parallel against the sequential version. As the work mentions, substantial speed-ups could be achieved with the GPU-based algorithm (a ratio of CPU vs. GPU time of up to 23.647 as the board size increased to 64 x 64).

– **In what concerns to best first searches, the A* algorithm** GPU parallelization is described in [15] applying techniques like:
  • Using multiple threads to work together on the same path implying shared memory and thread synchronization;
  • Partitioning of the search space for each thread to compute, also known as hierarquical breakdown, which consists in selecting entry and exit points of each part, executing small A* searches between those points and finally running a global A* search on an abstract graph containing the parts as atomic elements.
– **In what concerns to RSA, GPGPU is applied:**
  • **On MCTS**[9] ([5],[13],[16] and [21]) which typically supports three ways of parallelization:
    ∗ 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).

  A schematic representation of the different types of parallelization used in MCTS can be viewed in Fig. 7.
  Another method was proposed in [13], 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.
  • **The R* algorithm** is a randomized algorithm based on A*. It constructs a high level graph by picking random states, executing small A* searches to those states, and executing a search on the high level graph. The proposed algorithm is GPU parallelized on [10] in a similar way than A*, as it divides the main graph in pivot points and then does parallel searches on partial paths.

[9] MCTS 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. For a more detailed explanation of this algorithm, the viewer can visit `http://www.cameronius.com/research/mcts/about/index.html`.

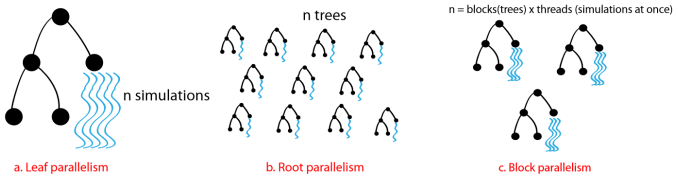

Fig. 7. Scheme of different MCTS parallelizations. Source: [16]

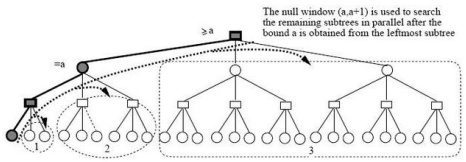

Fig. 8. Scheme of PVS and thread paths[10].

2.3 Racing Videogame AI

A brief description of racing videogames AI history helps the reader understand the importance and the meaning of some terms mentioned in this proposal. This Section is also going to introduce some approaches applied to construct TORCS controllers, as some aspects like the methods used to test the controller or to represent the states can be analyzed and considered for this work.

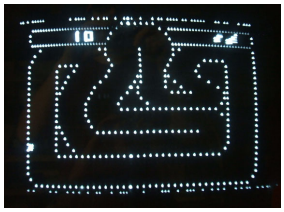

Fig. 9. Screenshot of Gran Track 10[11].

History of Racing Videogame AI The history of racing videogame AI is very interesting. Little people know that the first racing games did not even had

[10] Source: `https://chessprogramming.wikispaces.com/Parallel+Search`
[11] Source: `https://indiegamerchick.com/page/2/`

AI built into them. One example is Gran Trak 10 [12] (see Fig. 9), a Single-player racing arcade game released by Atari in 1974.

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. But, as more computing power became available, racing games started to use path-finding and decision making algorithms, because not only the former technique cannot be applied to open-world racing games (Recent open-world racing games include Insane games by Codemasters [13] [14] or Burnout Paradise by Criterion Studios [15]), but also the fact that Racing Line methods on their own do not work well when there is a big number of cars following the same line, as in most serious simulators (for example, in an endurance race as Le Mans [16] there can be more than 60 cars at the start!). Other methods have been used for IA decision making such as neural networks in Colin McRae's Rally 2 by Codemasters [17] or learning by imitation in Forza games by Microsoft Studios [18].

Other Approaches on TORCS Controllers A wide variety of techniques have been applied to TORCS AI, some more simple, some more complex, but the interest resides in the ones that were scientifically documented and/or were used during actual competitions.

Some approaches appear to be based on classical Racing Line methods, like the "bt" driver already available in the TORCS distribution 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 approaches have been developed and documented:

- **Inverse Reinforcement Learning** [6] was applied to a TORCS controller with the objective of testing a framework for autonomous road driving.
  The state representation included features like the lateral displacement with respect to track center, 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 and timesteps. 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.

[12] http://www.arcade-museum.com/game_detail.php?game_id=7992
[13] http://1nsane.invictus.hu/html/
[14] http://store.steampowered.com/app/35320/
[15] http://store.steampowered.com/app/24740/
[16] http://www.lemans.org/en/24-hours-of-le-mans/
[17] http://www.ai-junkie.com/misc/hannan/hannan.html
[18] http://news.xbox.com/2014/09/30/games-forza-horizon-2-drivatars/

A motion prediction algorithm was implemented. It was described like: (It) "*receives tracking data (ie position, orientation and velocity) and outputs H grids, representing the posterior probability of the space being occupied at times{t1,···,tK} in the future*".
The cost function parameters were estimated by the learning algorithm. They were based on exhibited behavior (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*, implemented as a Markov Decision Process, which induced the required gear, acceleration and break commands in order to keep nominal velocity;
- A *pose controller*, which sought the points, returning an angle capable of maintaining nominal position and orientation.

– **MCTS** [8] was applied to a controller in order to race the car around the track as fast as possible. This article is interesting because of not only the world representation but also the testing method used.
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 those states. The evaluation function used is displayed in Equation (1).

$$k = (d - d_0)/(1/2 * a_{max} * t^2 + v_0 t), \quad k_\perp = k^2 * P \tag{1}$$

where d is the distance along the track line in the 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 for the purpose of the work 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 method used to test and compare this approach was the distance the car drove in a determined amount of time (10000 game ticks). 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.

Vehicle PID Controller When the search is based on inexact information, a commonly applied error correction technique is the *Proportional–Integral–Derivative* (PID) method. This method is described in this section as its use can be considered for this work *if state simulations are not exact predictions of next states.*

PID can be defined as an error correcting technique which is commonly used in industrial control systems. The objective of the technique is to minimize the error over time, which gets practical action results closer to the theoretical predictions. The error is found by subtracting the process variable (PV) to the theoretical set point (SP). The action taken by a mechanism that implements this type of controller is affected not only by present error (the proportional component), but also the past errors (the integral component) and future error predictions (the derivative component).

All in all, the final error is given by a weighted sum of the three components described above. The correspondent equation can be seen in Equation (2).

$$u(t) = K_p e(t) + K_i \int_0^t e(t)\delta t + K_d \frac{de(t)}{dt} \tag{2}$$

In Equation (2), $e(t)$ is the error found, $u(t)$ is the variable to be minimized (the control variable) 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 pre-calculated or dynamically adapted (in that case the controller is called an *Adaptive PID Controller*) in order to make the system react properly in its environment. Use cases for this method are in the stabilization of a rocket or in vehicle trajectory following.

The article [12] refers how to generally implement and tweak a PID controller so that it can adapt to a racing environment. The reader can understand how each weight interacts with the actions taken, the errors that generally occur and how to fix those errors. As cited in the paper, 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;
2. Increase the K_i weight to fix the steady-state error (the steady state error occurs when the controller settles in a state deviated from the desirable);
3. Adjust K_d to reduce overshoot and settling time.

Complex controller applications in racing are expressed. The ones related to this work are:

- *Types of Steering Controller*, where the author 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;
- *Speed controller - split channels*, where two advanced techniques are introduced for controlling the speed: *throttle only* and *exact tracking*. While the *throttle only* approach slows the car just by taking the foot of the gas pedal, the *exact tracking* breaks for every speed correction. The *throttle only* approach can be used only for small negative errors. The other approach can be used when the negative errors are relatively high.

All in all, a more realistic speed control can be implemented using the two techniques above, which can help in the development of a more suited controller for this work's bot.

2.4 Discussion

The inspiration for the approach presented in this work was the description of the RRT algorithm explored in [2] and [18], and used in [17]. After observing the benefits of using the RRT*, a decision was made to analyze the improved version.

At a first glance, it is concluded that the original RRT is better suited for parallelization than its improved version (as mentioned in Section 2.1), because it lacks the rewiring and rearranging phases which reveal interdependence. Nevertheless, there is a possibility of deviation from the plain original algorithm.

Besides that, as this particular game genre has non holonomic constraints like $\dot{x} = v(cos\theta\hat{x} + sin\theta\hat{y})$ implied, the use of the RRT algorithm family seems to be adequate.

Even more important is the fact that this algorithm is majorly applicable to kynodinamic problems (velocity, acceleration and/or force bound), which are obviously related to the type of problems approached in this work.

TORCS was chosen as the case to study because of the importance it represents to the development of AI techniques through the creation of bots. It has got to be remembered that besides the game was created in the 90's (It is considerably old), competitions are still held to put new bot ideas to the test. Unfortunately, some more sophisticate ideas like perhaps the one in this work commonly have not got the opportunity to compete as the competition enforces some prohibitory rules about track information acquaintance during the race.

The fact that other RSA like the MCTS proved good results on TORCS also captivated the interest to develop this work.

The evaluation function present in [8] (seen on Equation (1)) will also be analyzed as it presents a good compromise between the distance traveled in a straight line and attenuation of that distance when driving on corners, where the acceleration is reduced along with the velocity (the denominator).

The state representation present in [8] can also serve as a base to this implementation. A description of the state representation considered for this proposal can be viewed in Chapter 3, in the Approach section.

The evaluation of the work cited in [8] which consisted, as mentioned before, in the measurement of the bots traveled distance in 10000 game ticks, can also be applied to this work because of its simplicity and effectiveness (it is a good way to see if the approach reveals to be faster than others).

An interesting fact about the work in [6] is that the library used had a GPU-based plug-in architecture, which indicates at least that TORCS can cope with a GPGPU accelerated algorithm. The purpose of this work is to build a racing controller which in theory will imply other types of demands, for example no drops in the frame refreshing rate while driving at high speed.

A PID controller can be considered to compensate the fact that in TORCS some of the information concerning the car is hidden to the controller interface (Driver class), namely aerodynamic and mechanical factors. If a workaround cannot be found, an exact forward model cannot be generated, because *a relation between the controls and the acceleration of the vehicle cannot be directly found.* The work in [12] can serve as a base for tweaking a bot's PID controller so that it responds effectively.

3 Solution Proposal

3.1 Approach

State Representation Like what generally happens in this kind of problem, the objective of each state is to represent a predicted player situation.

Thus, the representation that seemed more adequate was including attributes like position, speed and acceleration. The pedals position percentage and steering angle can be considered if control decisions need to support the planning phase. These attributes, which allow to control and to gather information about the car, can be obtained from the car's API structure in TORCS (the *carElt* structure). The initial/final speed, acceleration and initial/final position of the car can also be used to test constraints or to check if the point is valid (if it lies on track for example). An overview of the state attributes can be consulted in Table 1 and the function of each one is clarified by the scheme provided in Fig. 10.

Search Parallelization As mentioned before, GPU parallel implementations using the RRT family can be found in [2]. Although they produced satisfactory results, the parallelization was based on the collision checking method, which is not directly related to the search. Nevertheless, constraint parallelization can be considered for this case by culling invalid RRT points but only after tree generation. The point culling procedure is massively keen to parallelization, because there are no dependencies between the culling of different points.

Concurrently generating sub-trees like in MCTS's leaf playout parallelization can help to improve the search space exploration by producing several solutions

Table 1. State Attributes

Var	**Min Value**	**Max Value**	**TORCS API Attr.**
Gas Pedal Pos (%)	0 (0%)	1 (100%)	*car._accelCmd*
Break Pedal Pos (%)	0 (0%)	1 (100%)	*car._breakCmd*
Steering Angle (in radians)	N/A (manually bound to *-car._steerLock*)	N/A (manually bound to *car._steerLock*)	*car._steerCmd*
Initial Car Position	N/A	N/A	*(car._posX; car._posY)*
Final Car Position	N/A	N/A	*(car._posX; car._posY)*
Initial Speed	0	N/A	$\sqrt{(car.\_speedX)^2 + (car.\_speedY)^2}$
Final Speed	0	N/A	$\sqrt{(car.\_speedX)^2 + (car.\_speedY)^2}$
Acceleration	0	N/A	*Final Speed - Initial Speed*

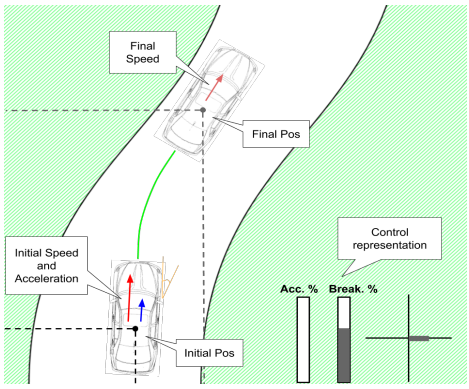

Fig. 10. Visual identification of the attributes in each state. The darker car represents the initial position, the red vector the car's initial speed, the blue vector the car's acceleration; the green line represents the theoretical trajectory; the lighter car represents the predicted final position and speed; Finally a visual telemetry represents the controls (pedals position and steering axis).

at a time. If search space division is applied, independent sub-trees are built in different portions of the search space. The best sub-tree paths can be compared and an overall best path deducted. This approach can also be considered because the decision quality and efficiency is directly related to the bot's behavior.

In this manner, the used parallelization can either be (1) directly related to the search if based on the node generation, (2) on constraint checking like in [2] or (3) a combination of both approaches. The methods of parallelization described here are represented by the scheme of Fig.11.

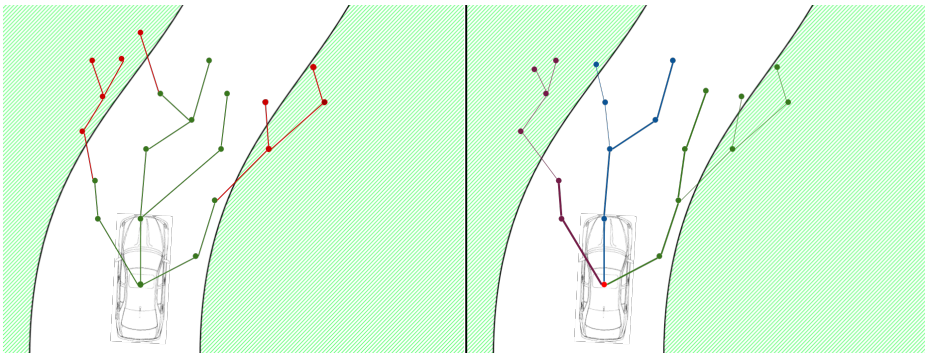

Fig. 11. Representation of the parallelization possibilities described on the Search Parallelization subsection. On the left scheme, the points culled for being or having a parent outside the track are colored red while the others are colored green. The sub-tree parallelization approach by space division can be seen on the right, where different colors represent sub-trees built by different threads on the initial node. The thick lines represent best sub-tree plans.

Global Contextualization On the global context, this work can be seen as an AI driver implementation different from the others created, as the use of GPGPU both to improve the AI behavior and display proposes was not yet documented as far as the author knows. Hopefully this technique can be used in the future to produce better AI for games.

3.2 Architecture

Tools The work is going to be developed in the C++ programming language, using Microsoft's Visual Studio[19] 2013 and CUDA's development toolkit [20] 7.5. Newer versions of the CUDA tool have not yet proven to be as stable and as well documented as the one mentioned above. As a result, the 7.5 version is preferred. The 2013 Visual Studio was chosen because the current version did not give support to the CUDA's toolkit at the installation time.

[19] `https://www.visualstudio.com/`

[20] `https://developer.nvidia.com/cuda-toolkit`

Modules The TORCS controller that is going to be developed will consist of several modules, similar to the approach in [17]. A schematic representation of the solution can be found in Fig. 12. It includes:

- **A planning module**, which will be composed by the sequential and GPU parallel versions of the search algorithm;
- **A control module**, which will coordinate the actions needed to drive the car, by calling the TORCS back-end;
- **The TORCS back-end**, which will provide the necessary game core procedures to describe and control the car and describe the track. It is possible to get additional car information as the gas tank state, the cars category, the relative position to the track edge or the current lap, time and distance from start lane.

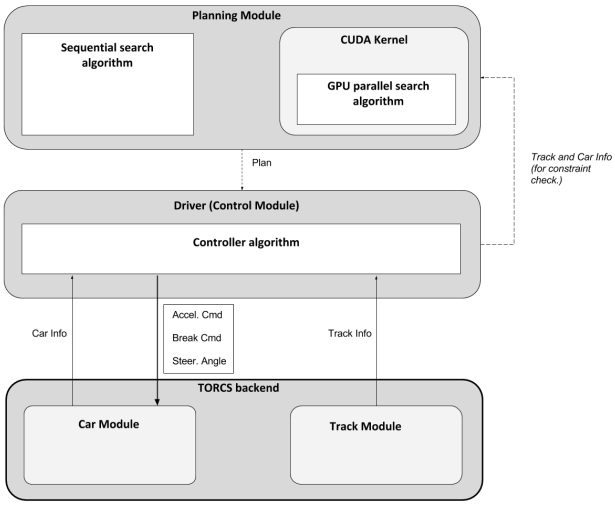

Fig. 12. Diagram of the solution proposed in this document for the TORCS bot. The several modules are displayed as different levels.

Preliminary Tests / Proof of Concept In this subsection, preliminary solutions developed alongside this proposal are described. The objective of these solutions is to demonstrate, at first glance, the feasibility of the work, or in other words, to serve as a proof of concept.

- **Cuda Test** Summing up, this solution consisted in a simple bot that sought map coordinates. In the same time, a CUDA kernel was built to be called in intervals of 300 game ticks, which generated a random state (composed of a random position, speed and acceleration).

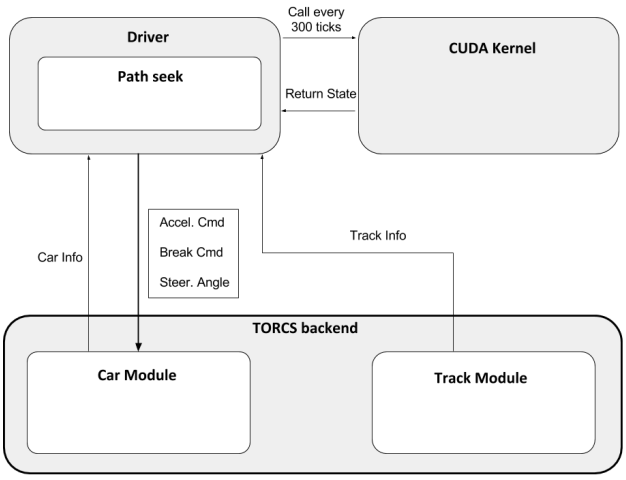

Fig. 13. Diagram of the preliminary CUDA testing solution used for proving the concept.

A scheme of the architecture of this solution can be viewed in Fig. 13.
The same graphical board ran both the kernel and the game. Although the game executed without errors, CUDA calls created screen tearing artifacts[21] which were reduced by changing the game's resolution;

– **Algorithm Test** A simple implementation of the RRT was also tested to check the feasibility of the approach. Using simple states to represent predictions, which contained positions, speeds and accelerations, and limiting the tree span to 30 segments ahead of the car, random trees were successfully built and plans were generated. Therefore a new search calculation was required every time the car reached the last point of the current search. The used evaluation function was the quadrance of the euclidean distance between points. The formula can be seen in Equation (3).

$$(x_{parent} - x_{curr})^2 + (y_{parent} - y_{curr})^2 \tag{3}$$

The plan path was found by backtracking the initial point's furthest state (considering the traveled distance along the center of the track). The bot then followed the plan path, state by state. Although the bot's behavior was inadequate due to poor control (too much cornering speed) and even casually random (as expected from a stochastic algorithm), the solution inspires a skeleton for the work proposed here. A screenshot of the algorithm's test execution can be seen in Fig. 14.

[21] Screen tearing artifacts are not desirable because they distort the image displayed. More about this topic can be consulted in `https://en.wikipedia.org/wiki/Screen_tearing`

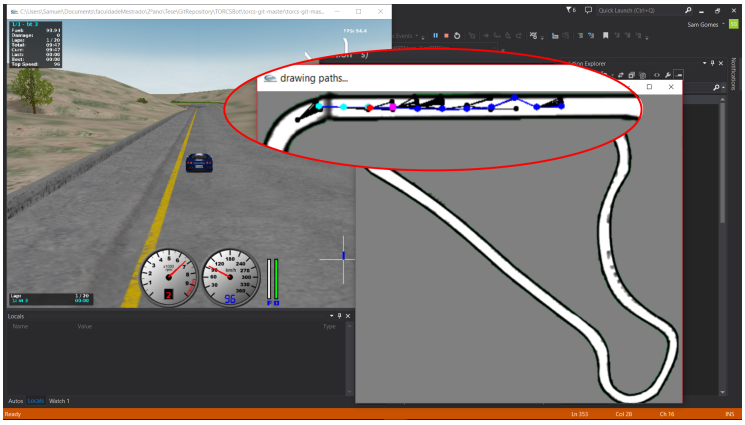

Fig. 14. Screenshot of the algorithm test execution. The generated RRT can be seen in black along with the best path (highlited in blue). The car position is represented by the red point. The purple point represents the next point in the path.

Hopefully, the execution success seen on preliminary tests can scale up to the development of the approach referred on this document.

3.3 Evaluation

Experimental Process Description The evaluation of the work will be divided in two phases:

- **Measure the elapsed time of one complete search procedure** of both the sequential and parallel implementations, testing the improvement achieved by the parallel one;
- **Compare the GPU parallelization approach with the sequential approach also checking the approaches already tested in [8], using as a metric the distance traveled in 10000 game ticks**. This aspect is approached because a time efficient search algorithm does not serve its purpose if the path constructed exhibits bad behavior[22] (a point is generated behind the car, for ex.).

Limitations of the Approach The big limitation of this approach is the possibility of creating graphical artifacts or other types of malfunctioning as a consequence of using the same GPU to process graphics and general information. This can happen when the GPU becomes overloaded through its use for general purpose computation, and as a result does not update the graphical pipeline buffers correctly.

The way the IDE will cope with CUDA compilation for the TORCS bot along the development of the work is also uncertain. Nevertheless, until now, good results were observed for this issue.

[22] The reader is advised to remember the definition of behavior given in Section 1.3.

3.4 Work Schedule

This work will include the bot development (practical component) alongside the writing of the thesis. The plan for the bot's development throughout the year will go as follows:

First, an AI driver will be built and tested for minimizing the lap times. This driver will run alone in the track. Then, dynamic obstacle avoidance will be implemented to make the driver cope with opponents, either human or non-human. This driver will first compete with one opponent and then scale up to several opponents.

All in all, the work phases can be described by the following list:

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;*

Applying the tasks enumerated above, a schedule for the development of the work can be seen through the Gantt chart in Fig.15.

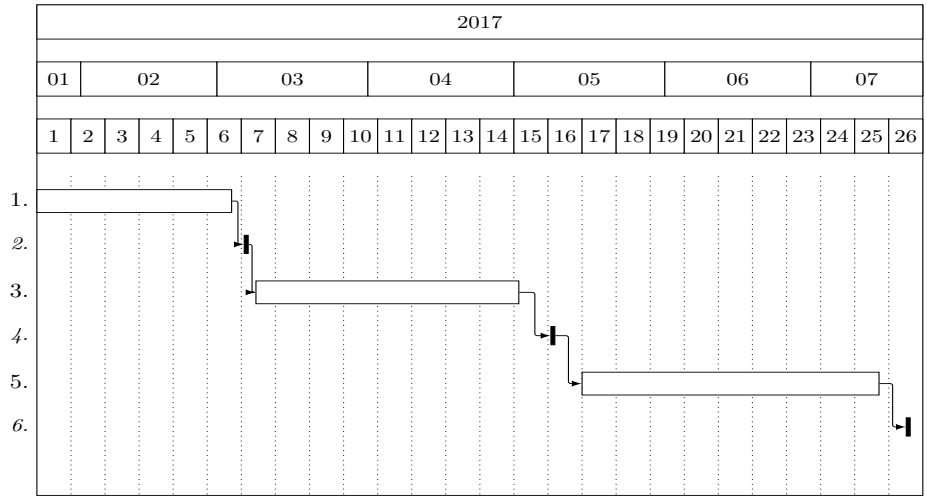

Fig. 15. Planning chart for the work here presented. The scale is divided by year, month and week.

Bibliography

[1] CUDA_C_Programming_Guide. Tech. rep., NVIDIA Corporation (2016), http://tinyurl.com/jps5swu

[2] Bialkowski, J., Karaman, S., Frazzoli, E.: Massively parallelizing the RRT and the RRT*. In: IEEE International Conference on Intelligent Robots and Systems. pp. 3513–3518. IEEE (2011)

[3] Botta, M., Gautieri, V., Loiacono, D., Lanzi, P.L.: Evolving the Optimal Racing Line in a High-End Racing Game. In: 2012 IEEE Conference on Computational Intelligence and Games (CIG'12). pp. 108–115. IEEE (2012)

[4] Cardamone, L., Loiacono, D., Lanzi, P.L., Bardelli, A.P.: Searching for the optimal racing line using genetic algorithms. In: Proceedings of the 2010 IEEE Conference on Computational Intelligence and Games, CIG2010. IEEE (2010)

[5] Chaslot, G.M.J.B., Winands, M.H.M., Van Den, H.J.: Parallel Monte-Carlo tree search. In: International Conference on Computers and Games Notes in Bioinformatics. pp. 60–71. Springer (2008)

[6] Dizan, V., Yufeng, Y., Suryansh, K., Christian, L.: An open framework for human-like autonomous driving using Inverse Reinforcement Learning. In: 2014 IEEE Vehicle Power and Propulsion Conference (VPPC). pp. 1–4. IEEE (2014)

[7] Ferreiro, A.M., García, J.A., López-Salas, J.G., Vázquez, C.: An efficient implementation of parallel simulated annealing algorithm in GPUs. Journal of Global Optimization (Springer) 57(57), 863–890 (2013)

[8] Fischer, J., Falsted, N., Vielwerth, M., Togelius, J., Risi, S.: Monte Carlo Tree Search for Simulated Car Racing. In: Proceedings of the Foundations of Digital Games Conference. ACM (2015)

[9] Gordon, V.S., Whitley, D.: Serial and Parallel Genetic Algorithms as Function Optimizers. Tech. rep., Colorado State University (1993)

[10] Kider, J.T., Henderson, M., Likhachev, M., Safonova, A.: High-dimensional planning on the GPU. In: Proceedings - IEEE International Conference on Robotics and Automation. pp. 2515–2522. IEEE (2010)

[11] Maxim Likhachev, Anthony Stentz: R* search. In: Proceedings of the National Conference on Artificial Intelligence (AAAI). ACM (2008)

[12] Melder Nic, T.S.: Racing Vehicle Control Systems using PID Controllers. In: Steve Rabin (ed.) Game AI Pro: Collected Wisdom of Game AI Professionals, chap. 40, p. 10. CRC Press, USA (2013)

[13] Mirsoleimani, S.A., Plaat, A., Van Den Herik, J., Vermaseren, J.: A New Method for Parallel Monte Carlo Tree Search. Computing Research Repository (2016), http://arxiv.org/abs/1605.04447

[14] O 'neil, M.A., Burtscher, M.: Rethinking the Parallelization of Random-Restart Hill Climbing A Case Study in Optimizing a 2-Opt TSP Solver for GPU Execution. In: Proceedings of the 8th Workshop on General Purpose

Processing using GPUs. pp. 99–108. ACM (2015), http://dx.doi.org/10.1145/2716282.2716287

[15] RAFIA, I.: A* Algorithm for Multicore Graphics Processors. Ph.D. thesis, Chalmers University of Technology (2010)

[16] Rocki, K., Suda, R.: Parallel Monte Carlo Tree Search on GPU. In: Eleventh Scandinavian Conference on Artificial Intelligence. pp. 80–89. IOS Press (2011)

[17] Soares, R., Leal, F., Prada, R., Melo, F.: Rapidly-Exploring Random Tree approach for Geometry Friends. In: Proceedings of 1st International Joint Conference of DiGRA and FDG. DIGRA (2016)

[18] Steven M. LaVelle: Rapidly-Exploring Random Trees: A new Tool for Path Planning. Tech. rep., Iowa State University, USA (1998)

[19] Strnad, D., Guid, N.: Parallel alpha-beta algorithm on the GPU. In: ITI 2011 33rd Int. Conf. on Information Technology Interfaces. pp. 571–576. IEEE, Croatia (2011)

[20] Uhlenbein, H., Augustin, S.: Parallel Genetic Algorithm in Combinatorial Optimization. Tech. rep., Universidad de Málaga (1992)

[21] Yoshizoe, K., Kishimoto, A., Kaneko, T., Yoshimoto, H., Ishikawa, Y.: Scalable Distributed Monte-Carlo Tree Search. In: Fourth Annual Symposium on Combinatorial Search. pp. 180–187. ACM (2011)