Application and Design of GPU Parallel RRT for Racing Car Simulation

Samuel Gomes, João Dias and Carlos Martinho

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 The Open Racing Car Simulator (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%.

Index Terms—General-Purpose computing on Graphics Processing Units, Randomized Search Algorithms, Iterative Parallel Sampling RRT, 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) [12], R* [8] (based on A*) or Monte-Carlo Tree Search (MCTS) [5]. In particular, RRT can adapt to rapidly changing worlds and/or high dimensional state spaces, dynamically generating the representation of such spaces.

To speedup 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 ([3],[11]), R* [7] and even the RRT itself ([2], [6]). 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.

Racing games can benefit from the use of RRT, as directional speed and steering generate a big continuous state space. 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?

To approach such aspects, a newly proposed version of RRT, named Iterative Parallel Sampling RRT (IPS-RRT) was parallelized through GPGPU and applied to The Open Racing Car Simulator (TORCS)[1]. TORCS is a car racing videogame extensively used for academic research. It consists of a multi-platform open source game which serves as a base for the construction of AI controllers.

2 Related Work

2.1 Existing Bots for TORCS

TORCS presents an API to use in the development of bots. Such API includes attributes which help to describe the car's state and the track.

Although, to our knowledge, no work applied RRT to TORCS, other approaches have been explored. An Inverse Reinforcement Learning bot [4] was developed with the objective of testing a framework for autonomous road driving. That approach involved two sub controllers: a speed and pose sub controllers. The division of the controls in attributes that control the

1. http://torcs.sourceforge.net/. (as consulted on Thursday 31st August, 2017).

speed (related with the pedals) and steering (related to the steering wheel) is very interesting and was applied to this work's controller.

An MCTS racing bot was also developed[5]. It used a state space based on physical aspects like positions and velocities. Such bot then applied that spatial representation to predict future states through a physics based forward model. This state representation approach revealed to be very straightforward and effective. Therefore, a physics based state representation was used in this work too.

2.2 Improving RRT Performance

Several techniques have been applied to improve the performance of RRT. RRT*, referred in [2], is an optimization of the RRT algorithm to asymptotically converge to better solutions. 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 less prune to parallelization than the plain RRT. Constraint parallelization [2] 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 and joining them to produce a global tree. Such tree can then be searched to find a solution. Another approach for parallelizing RRT is presented in [6], Bulk Synchronous Distributed RRT (BSD-RRT). In such algorithm, processes create several RRT samples, updating a global tree through message broadcasting. Approaches such as the ones in [6] revealed to be applied in distributed computation contexts.

2.3 About GPGPU

Circuit wise, a Graphical Processing Unit (GPU) is a very different chip from a Central Processing Unit (CPU), because the purpose is completely different. While CPU's Arithmetic and Logical Units (ALUs) are built to deal with complex operations, a GPU deals with a massive number of simple operations executed at the same time. As a result, GPGPU implies a specific programming model. That model is mapped by toolkits such as CUDA[2] or Open CL[3].

2. http://www.nvidia.com/object/cuda_home_new.html (as consulted on Thursday 31st August, 2017).

3. https://www.khronos.org/opencl/ (as consulted on Thursday 31st August, 2017).

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. Threads in the same block can share data through shared memory and synchronize their accesses through synchronization methods.

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 memory hierarchy and cannot directly access main memory. Such memory hierarchy is detailed next:

- Each thread has its local memory, each thread block has its shared memory and all blocks can access a global memory. Global memory is associated with the VRAM and therefore has high latency (as referred in [1] about 300 clock cycles). Shared memory is used when the data can be divided into chunks. It is associated with the GPU's cache and therefore has low latency (as referred in [1] about 11 clock cycles);
- Additionally, constant and texture memories exist in the VRAM but are cached and therefore more efficient than global memory. Constant memory is normally used when data is highly concurrently read. Texture memory is normally used when data is randomly accessed a big number of times. However, as it presents a different manipulation API in comparison to the other types of memory, its adaptability to certain problems is harder.

  Although being more efficient, shared and constant memories have a common disadvantage: they are much smaller than global memory[4].

2.4 Maintaining a Smooth Framerate

Maintaining a smooth 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. This work focuses on the TORCS videogame which is frame independent. Therefore, 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. A good benchmark for the time that can be spent on simulation interruptions is ≈ 0.0417 of a second per frame (24 frames per second), as below this rate low rate artefacts can start to be notorious on modern devices.

3 Iterative Parallel Sampling RRT

After analysing existing RRT parallelization approaches, it was concluded that a more suitable one was needed.

4. While the size of global memory is limited by the graphics card VRAM, the size of constant memory is limited to around 64KB and the size of shared memory to around 48KB.

Existing approaches were either too simple for this study [2] or applied to distributed contexts instead of in parallel ones [6]. This chapter describes a sub product of this work, which is a variant of RRT named Iterative Parallel Sampling RRT.

Instead of applying constraint parallelization (like in [2]) 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(nIterations,nParSamples,T)
2:      loop from 0 to nIterations:
3:          launch nParSamples threads computing this:
4:              $T' \leftarrow T$
5:              $x_{rand} \leftarrow randomState()$
6:              $x_{near} \leftarrow nearestNeighbor(x_{rand}, T')$
7:              $x_{new} \leftarrow applyDelta(x_{rand}, x_{near})$
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:         $syncronizeThreads()$ , $T \leftarrow$ 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[5] 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 [6] 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 threads generate only one sample between synchronization points and a process in BSD-RRT generates several samples between a tree broadcast. 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 the full extent of the search space.

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 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. 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. 1 f).

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

The developed TORCS bot consists of several modules. A schematic representation of the solution can be found in Fig. 2. It includes:

- **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 the 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 plan or a time limit is reached. When the

5. By thread synchronization it is meant that the threads wait for the slowest one before the execution continues.

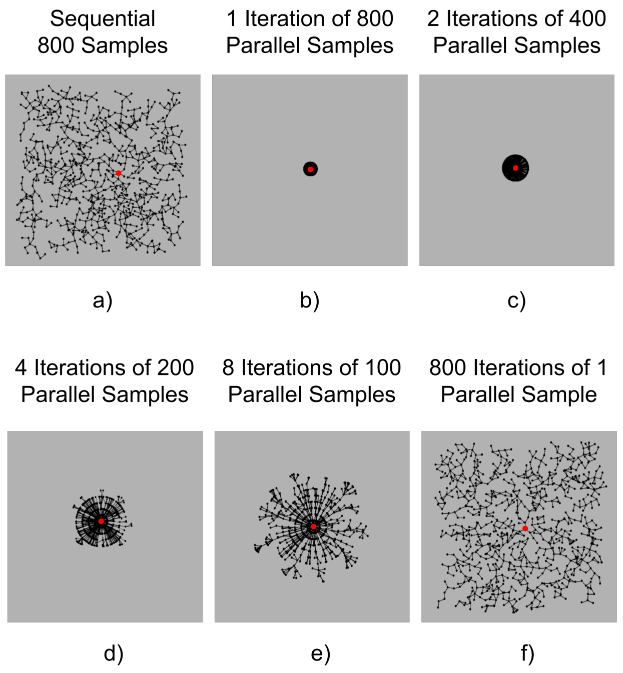
Sequential 800 Samples
1 Iteration of 800 Parallel Samples
2 Iterations of 400 Parallel Samples
a)
b)
c)
4 Iterations of 200 Parallel Samples
8 Iterations of 100 Parallel Samples
800 Iterations of 1 Parallel Sample
d)
e)
f)

Fig. 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.

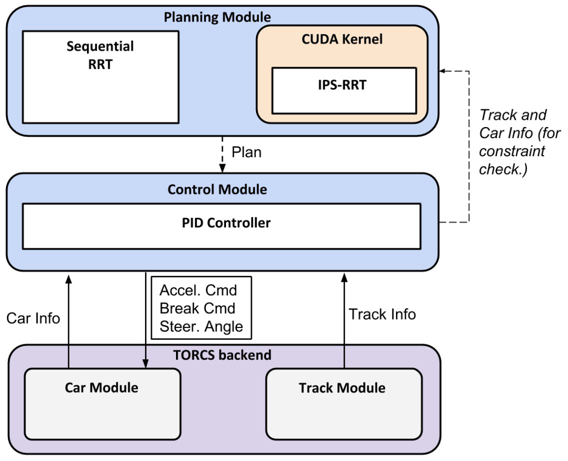

Fig. 2. Diagram of the TORCS bot's architecture and modules.

planning module returns a new plan, the control module executes it. This procedure is repeated until the end of the race.

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).

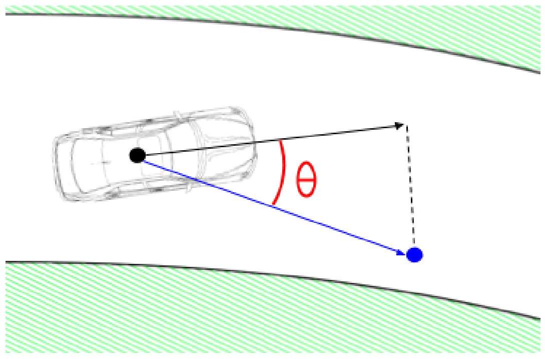

Fig. 3. Used method to determine the steering angle. The target search point is represented in blue. The position of the car is displayed in black along with the car's direction vector. The angle between the car's direction and the current state position, displayed as θ, is obtained by the arc-tangent of the legs of the triangle.

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 [4]). The pedals are controlled by a Proportional–Integral–Derivative (PID) controller[6] as the relation between their position and the car's acceleration varies from car to car and cannot be directly acquired. Nevertheless, 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 determined as described in Fig. 3.

4.2 RRT and IPS-RRT Applied to TORCS

As described in subsection 4.1, the planning module includes IPS-RRT as well as the plain RRT. Both algorithms represent the search tree as an array of states. It is important to notice that there is an implementation difference between IPS-RRT applied to TORCS and the pseudocode presented on section 3: threads do not keep copies of the main tree. Instead, they read from a shared copy of the main tree built at the start of each iteration.

Next, the implementation of the IPS-RRT methods (which include the RRT methods) is presented. 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} [7]. To calculate x_{new} (line 7), a set of intermediate states are generated between x_{near} and x_{rand}. Such states

6. The article [9] introduces PID and refers how to generally implement and tweak a PID controller so that it can adapt to a racing environment.

7. The angle variation between two velocity vectors is the minimal angle between them.

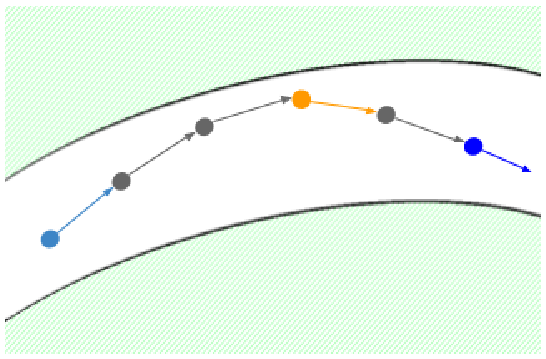

Fig. 4. Intermediate states and *delta* representation. x_{rand} is represented in dark blue and x_{near} in light blue. The states' velocities are also depicted accordingly. Intermediate states are represented in grey. x_{new}, represented in orange, is determined by choosing one of the intermediate 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* variation in RRT.

are computed by interpolating their velocities. Fig. 4 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[8] 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[9]. The best state is then backtracked to the initial state and the corresponding path is returned as the solution.

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 the global tree after a certain number of iterations.

5 CUDA Implementation Optimizations

After some preliminary tests, it was observed that the CUDA implementation was executing very slowly, getting much bigger search times than the CPU version, even while running thousands of threads. In order to decrease search times, the code was profiled and carefully analysed by running an IPS-RRT 800 state search in the GPU with only one thread and comparing it with the sequential RRT. While the CPU was executing such test in a few centiseconds, the GPU was taking around 5 to 6 seconds.

First, constraint checking was optimized in order 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. 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. Therefore, fewer segments are tested as the search stops when the tested state's segment is reached. After applying this improvement, the execution time of the CUDA implementation test presented above was decreased to about 2,5 to 2,7 seconds.

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. Because of such fix, the search time was reduced to about 1 second.

Next, it was concluded that some improvements could also be implemented in the nearest neighbour calculation, namely by using constant memory to store a simplified version of the main tree. That simplified tree would then be quickly traversed. As constant memory is read-only, the tree was copied from global to constant memory at the start of each iteration. A global tree was still required to preserve the full tree state between iterations. After implementing this optimization, the search time was reduced to about 0.7 seconds. This reduction comes from the fact that the GPU's cache has much lower latency than global memory (11 compared to around 300 clock cycles).

Finally, as copies from RAM to the GPU's global memory and from global memory back to it are slow, the tree and segments arrays were copied from RAM to the GPU's global memory in track loading time, instead of executing that copy each time the search was called. The only thing that had to be done in each search call was making sure to re-initialize the tree in order to clean previous data. 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.

6 Evaluation

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. 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;

8. 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.

9. 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.

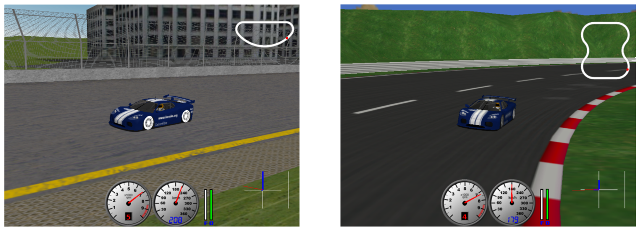

Fig. 5. Screenshots of the bot's execution in some of the test tracks.

- 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.

The work was developed in the C++ programming language, using Microsoft Visual Studio[10] 2013 and CUDA development toolkit 7.5 [11]. This combination was chosen because despite more recent ones being available, it was the most stable and well documented when the development started.

The used (laptop) graphics card was an Nvidia GTX 960M with 4GB of VRAM. 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 (3.6GHz of maximum turbo frequency) and 6 MB of cache. This CPU has 4 cores.

6.1 Studying the Graphics / Search Interaction

The first test consisted on executing 40 searches at the start position of the track "CG Speedway number 1". 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 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. Because the results were similar, only the 800 and 6400 states charts are presented (the extremes).

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.

10. https://www.visualstudio.com/ (as consulted on Thursday 31st August, 2017).

11. https://developer.nvidia.com/cuda-toolkit (as consulted on Thursday 31st August, 2017).

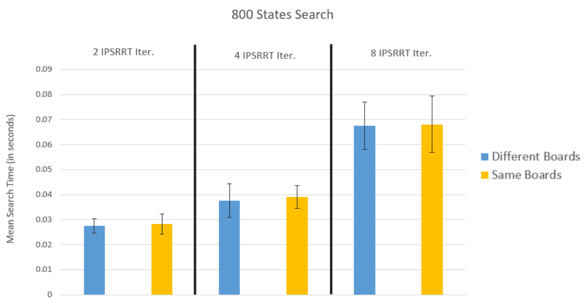

Fig. 6. Chart comparing the mean search times of GPU searches ran alongside graphics computations for 800 state searches.

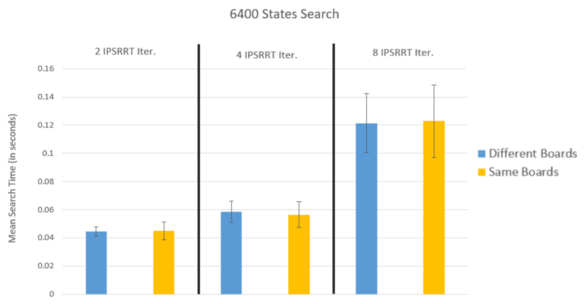

Fig. 7. Chart comparing the mean search times of GPU searches ran alongside graphics computations for 6400 state searches.

6.2 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 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.

6.2.1 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:

- **Number of Iterations Test**: In this test, each search generated 800 states, applying increasing number of iterations, from 1 (800 parallel threads) to 800 (1 thread). 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 chart of Fig. 8, 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, as they are inversely correlated. It can also be noticed that beyond 4 iterations, the search times

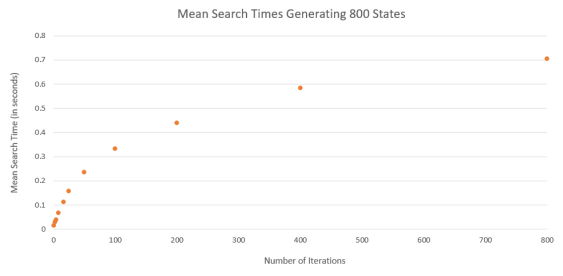

Fig. 8. Chart comparing the mean search times of an 800 state search with different number of iterations.

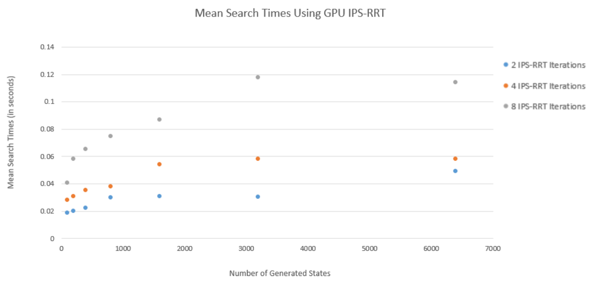

Fig. 9. Chart comparing the mean search times of the GPU implementation of IPS-RRT with different number of states.

escalate above the perception threshold discussed in subsection 2.4 (≈ 4.017 seconds);

- **Number of States Test:** In this test, each IPS-RRT search generated an increasing number of states, from 100 to 6400 (chart of Fig. 9). For comparison purposes, the same test was also run for the sequential implementation (chart of Fig. 10). The test was ran for 2,4 and 8 parallel iterations.
  The chart of Fig. 9 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, as expected, the sequential version (chart of Fig. 10) appears to scale much faster. 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 GPU 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 (≈ 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.

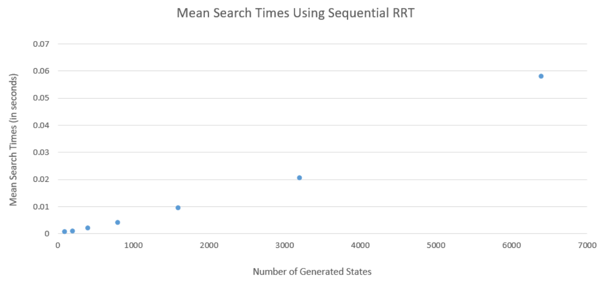

Fig. 10. Chart comparing the mean search times of the CPU sequential implementation with different number of states.

6.2.2 *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 running the bot for several runs of 10000 ticks (roughly 3 minutes and 26 seconds of in-game time), a typical TORCS competition qualification.

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 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. 11. 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. However, the big standard deviation lowers the significance of the test.

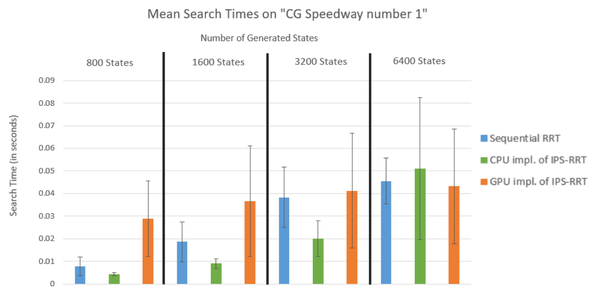

Fig. 11. Chart displaying the mean search times for the track "CG Speedway number 1".

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. 12. 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 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 the last paragraph of subsection 4.2) 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.

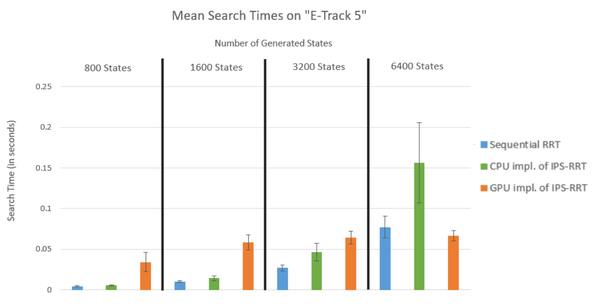

Fig. 12. Chart displaying the mean search times for the track “E-Track 5”.

Finally, the bot was executed for 10 runs of 10000 ticks on the track “Michigan Speedway”. The recorded search times can be consulted in the chart of Fig. 13. 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 this track has fewer segments to test in constraint checking. Therefore, the data quantity was not enough to overwhelm the CPU's cache.

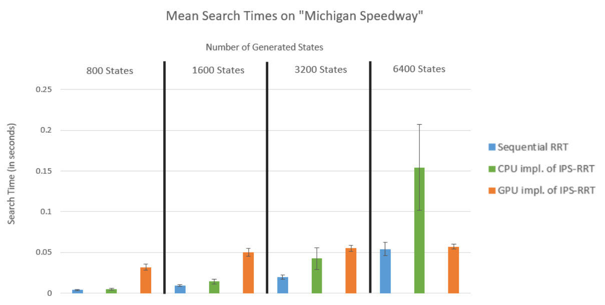

Fig. 13. Chart displaying the mean search times for the track “Michigan Speedway”.

6.3 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 mean lap times while executing the bot in the same circumstances as in the tests of 6.2.2. The minimum lap times were also acquired, but because the results were identical, they were not included.

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. 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 applied. 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 better neighbours for each state.

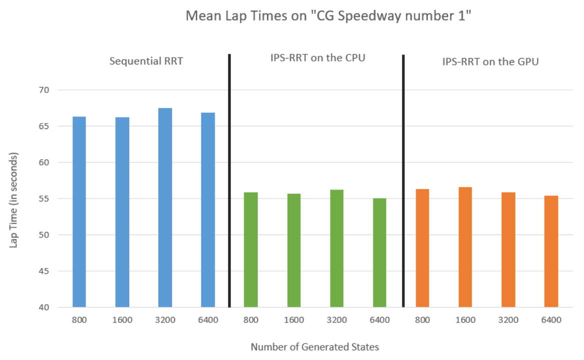

Fig. 14. Chart displaying the mean lap times for the track “CG Speedway number 1”.

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. 15. In this case, there was a 22% mean lap time reduction when IPS-RRT was applied. Maybe the bigger gain here is due to the fact that the parametrization was pushing the car so far that minor mistakes helped to inflate 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.

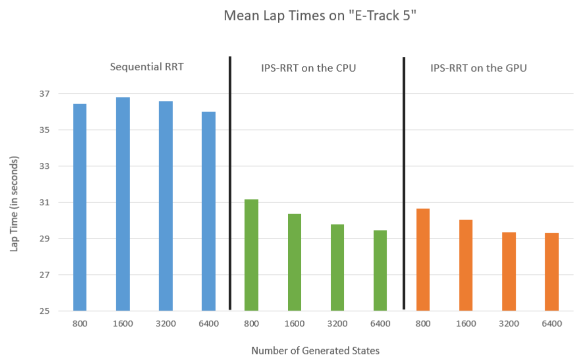

Fig. 15. Chart displaying the mean lap times for the track “E-Track 5”.

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. 16. In this case, there was just a 15% mean lap time reduction when IPS-RRT was applied. 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”.

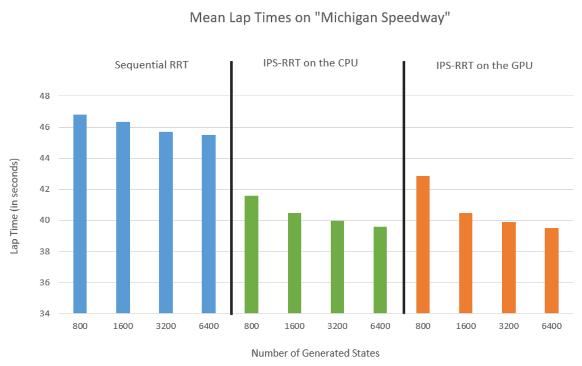

Fig. 16. Chart displaying the mean lap times for the track “Michigan Speedway”.

7 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. Nevertheless, with higher graphical demanding (more recent) games, the graphics rendering can be negatively influenced, especially when the GPU’s resources become nearly exhausted. This problem can also be attenuated through the pass of time, as GPUs have been evolving at a fast pace.

The GPU parallel IPS-RRT scales better than the sequential RRT, because of the influence of parallelism. Oppositely, the CPU implementation of IPS-RRT seemed to increase its search times very fast because of the copies it makes from/to partial trees. Moreover, 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 and because the presented time spans are very small.

It is 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 seconds) 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.

With 6400 states, GPU IPS-RRT reveals to be more computationally efficient than the sequential RRT. However, 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 phenomena at this track. Nevertheless, in “E-track 5” and “Michigan Speedway”, the search times were not influenced by empty searches.

As can be seen in the charts presented in Figs. 14, 15 and 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. Like mentioned before, 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. 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.

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 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. Important aspects about this approach were learned. They allow to answer the questions presented in section 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, although the same may not be verified when using highly graphical demanding applications such as recent videogames. Nevertheless, as GPUs are evolving at a fast pace, more and more resources become available.
  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, 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 4.2), except when a high number of states were generated. It was seen that such comparison is 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 hard to effectively apply them[12];
- Finally (and more importantly), IPS-RRT achieved a noticeable increase on the quality of returned solutions (overall reduction of around 19% in lap times)[13]. This proves that GPGPU-based implementation 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.

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. 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 7).

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 [5]. 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.

12. 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 IPS-RRT iterations.

13. 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 4.2).

References

[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] 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)

[4] 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)

[5] 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)

[6] Jacobs, S.A., Stradford, N., Rodriguez, C., Thomas, S., Amato, N.M.: A scalable distributed rrt for motion planning. In: Robotics and Automation (ICRA), 2013 IEEE International Conference on. pp. 5088–5095. IEEE (2013)

[7] 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)

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

[9] 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)

[10] Plaku, E., Bekris, K.E., Chen, B.Y., Ladd, A.M., Kavraki, L.E.: Sampling-based roadmap of trees for parallel motion planning. IEEE Transactions on Robotics 21(4), 597–608 (2005)

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

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