CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

High-performance trajectory planning: A GPU-acceleration performance study

A comparative performance analysis between a single-core, multi-core and GPU-accelerated trajectory planning algorithm

Master's thesis in Computer science and engineering

HUGO MÅRDBRINK
SIMON ENGSTRÖM

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2025

Master's thesis 2025

High-performance trajectory planning: A GPU-acceleration performance study

A comparative performance analysis between a single-core, multi-core and GPU-accelerated trajectory planning algorithm

Hugo Mårdbrink
Simon Engström

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2025

High-performance trajectory planning: A GPU-acceleration performance study
A comparative performance analysis between a single-core, multi-core and GPU-accelerated trajectory planning algorithm
Hugo Mårdbrink
Simon Engström

Supervisor: Miquel Pericas, Department of Computer Science and Engineering
Advisors: Ankit Gupta, Ivo Batkovic, Hannes Eriksson of Zenseact
Examiner: Miquel Pericas, Department of Computer Science and Engineering

Master's Thesis 2025
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2025

High-performance trajectory planning: A GPU-acceleration performance study
A comparative performance analysis between a single-core, multi-core and GPU-accelerated trajectory planning algorithm
Hugo Mårdbrink
Simon Engström
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Automated driving technologies and advanced driver assistance systems (AD/ADAS) have been a popular research topic since the automotive industry started pursuing software-defined vehicles. An instrumental part of AD/ADAS is the trajectory planning algorithm, which decides the trajectory for the given traffic environment. In recent years, trajectory planning algorithms have improved in both run time and trajectories.

While the algorithmic improvements have been apparent, there has been a lack of research on the suitability of parallelization and graphical processing unit (GPU) acceleration. Targeting the GPU is also highly relevant due to the increase of GPUs in a vehicle's computer architecture. This paper implements a spline-based trajectory planning algorithm in C++ for a single-core central processing unit (CPU), multicore CPU, and GPU-accelerated platform. Implementations were tested on a relevant automotive computing platform for accurate comparisons in a realistic scenario.

Ultimately, this thesis concludes that the GPU-accelerated implementation is better in every aspect measured and, in some cases, achieves a speedup of 2 to 3 orders of magnitude. Due to the much higher throughput, more solutions could be generated in a real-time scenario, leading to safer trajectories overall.

Keywords: GPU-acceleration, Parallelisation, Trajectory planning, Trajectory planning algorithms, Optimisation

Acknowledgements

First and foremost, we would like to thank Miquel Pericàs, our Chalmers supervisor, for his unique perspectives, deep understanding of the HPC field, and persistent guidance. We would also like to thank our Zenseact supervisors, Ivo Batkovic, Ankit Gupta, and Hannes Eriksson, for their invaluable domain knowledge and insight in trajectory planning.

Hugo Mårdbrink, Simon Engström

Gothenburg, 2025-06-17

Contents

List of Figures

List of Tables

1

Introduction

Autonomous Driving and Advanced Driving Assistance Systems (AD/ADAS) are systems used in vehicles to assist the driver, improve traffic safety, and mitigate the risk of human error in traffic scenarios. Some examples of these systems include anti-lock breaks, electronic stability control (ESC), and adaptive cruise control. Over the years, these systems have become more sophisticated with the digitalisation of vehicles and increased computation power. A subset of these systems must process large amounts of data to make decisions or help the driver make decisions while driving. For example, by monitoring the driver's steering patterns, the system can identify when the driver is tired and should take a break to rest. Furthermore, using sophisticated collision detection systems, the vehicle can emergency break if the system deems collision inevitable.

1.1 Problem statement

A central component in AD/ADAS systems is the trajectory planning algorithm that calculates optimal trajectories with the help of cost functions. For a vehicle to be able to adapt to an ever-changing traffic environment, both external factors and driver decisions need to be accounted for in the algorithm. One algorithmic apprach is finding the optimal trajectory from an initial state to a pre-defined final state [1]. In this case, optimal means that it has a minimal cost defined by cost functions that evaluate trajectories based on several metrics. These approaches usually work well in calm traffic environments. Still, in situations where obstacles can sporadically appear, and the driver can randomly change their intentions, the algorithm needs to be able to consider alternative trajectories to different end states in real time. The algorithm presented by Werling et al. [1] uses an iterative approach where multiple trajectories are generated for different driving strategies. Driving strategies refer to driving actions such as stopping and velocity keeping. This approach enables the algorithm to react better to changing environments and driver input.

With time, more driving strategies might be added to account for more traffic scenarios. More driving strategies infer an increased computational load that may become unfeasible for a sequential approach. Therefore, parallelism must be considered to reduce the algorithm's execution time. Furthermore, due to the simple nature of the calculations performed in the algorithm, the potential benefits of GPU-acceleration should be evaluated to determine the optimal parallelism approach

for GPU-accelerating the algorithm.

In recent years, extensive research has been done on developing algorithms for faster execution and more refined trajectories in real-time scenarios [2], [3]. To tackle the computational challenge of finding trajectories and evaluating them in real-time, McNaughton et al. [3] implemented their algorithm on a GPU, allowing them to yield faster results. The results of this implementation were promising, highlighting the potential of using GPUs to parallelise this type of workload. However, a comprehensive performance analysis is not present, raising questions about resource utilisation and the feasibility of executing the algorithm on vehicular GPUs. Furthermore, Werling et al. [1] claims that their algorithm is entirely parallelisable and that multicore execution would further reduce the execution time. This claim did not, however, have an attempted parallel implementation or performance review.

1.2 Research goals

This thesis's research mainly investigates how GPU-accelerating a trajectory planning algorithm affects different performance metrics. More specifically, the implementation to be evaluated is the trajectory planning algorithm proposed by Werling et al. [1]. The primary justification for this goal is to present the lack of performance analysis and performance and quality trade-offs within the trajectory planning algorithm domain.

One goal is to determine the speedup of accelerating the algorithm on a GPU relative to a CPU. The problem is answered in execution time and trajectory throughput for both GPU and CPU. Comparing the different metrics will give a relative speedup between the processing units and provide a conclusive answer as to which is faster and by how much. Furthermore, the trajectory planning algorithm itself leaves room for parameter adjustments, such as the number of traffic objects. These parameters are not static, and measuring different parameter configurations would conclude performance trade-offs when varying them.

Another goal is how the quality of trajectories is affected by offloading the computations to a GPU. It is necessary to conclude if the quality of the trajectories suffers or improves.

This thesis also aims to give a performance analysis of the GPU-acceleration of the algorithm, highlighting bottlenecks and opportunities for further optimization. This provides useful pointers for further research regarding GPU-acceleration of trajectory planning algorithms.

- **RQ1** How does parallelisation affect the execution time of the algorithm?
- **RQ2** Could parallelism increase the quality of the trajectory generation?
- **RQ3** How efficiently does the GPU-accelerated program kernels utilise the hardware in terms of compute and memory resources?

2

Background

The background gives details about the algorithm to understand optimisation efforts and implementation details. This chapter also provides theory about computer architecture and optimisation strategies, especially about GPUs.

2.1 Trajectory planning algorithm

Trajectory planning algorithms aim to find a path from one point or state to another, satisfying a set of conditions or minimising a cost calculated from a set of path characteristics. The algorithm this paper is based on by Werling et al. [1] is based on an optimal control problem.

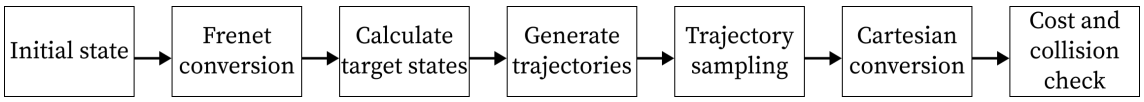

Figure 2.1: Steps of the algorithm

Figure 2.1 shows a simplified flow chart of the algorithm presented by Werling et al. [1]. In this case, the initial state refers to the initial state of the road. This traffic scenario gets converted into the frenet frame, and several target states are calculated. Trajectories are generated and sampled from these targets before being converted into cartesian coordinates. Lastly, collision checks and cost functions are run, and the winning trajectory is chosen.

2.1.1 Frenet frame

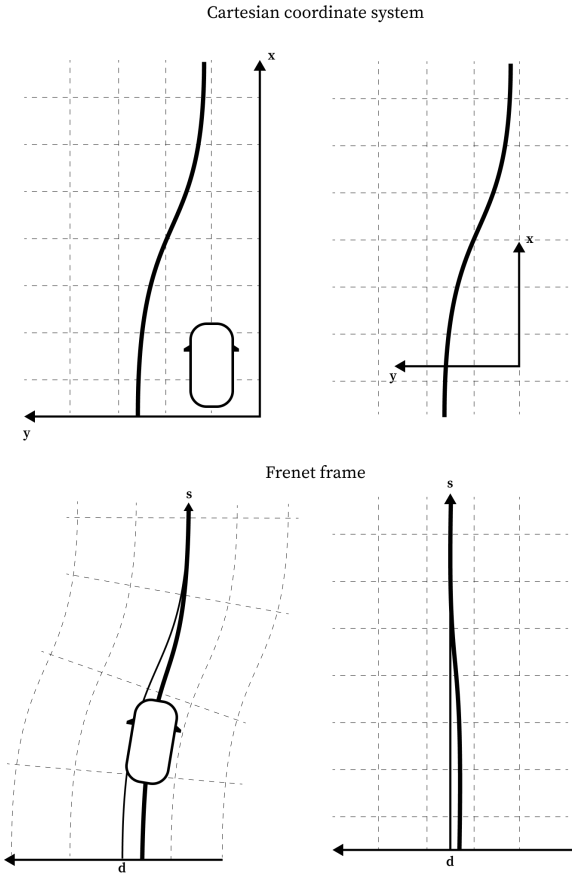

Figure 2.2: Cartesian coordinates and the frenet frame comparison

The frenet frame is a moving coordinate system defined by a curve's tangent, normal, and binormal vectors, which describe the curve's local orientation and curvature at each point [4]. In this case, the curve is the reference path following the road. A visualisation of a reference path converted to the frenet frame compared to a cartesian coordinate system can be seen in Figure 2.2. While a cartesian system is represented as x and y, the frenet frame is defined as s and d, where s is the longitudinal position and d is the latitudinal position [4].

2.1.2 Trajectory generation

The algorithm's trajectory generation part generates possible trajectories for specific targets. These targets vary primarily based on vehicle mode, giving them different properties like target time and velocity. The four driving strategies defined in the implementation are velocity keeping, stopping, lane change, and lane keeping. For each mode, trajectories are generated with lateral and longitudinal offsets [1].

$$N_{Trajectories} = N_{Modes} * N_{LatOffsets} * N_{LonOffsets} * N_{TargetTimes} \tag{2.1}$$

```
for (mode : vehicle_modes) {
    for(d_min..d_max) {

        // Combine polynomials
        for(0..target_time) {
            // Generation for each point
        }

        cost = totalCost(trajectory)
        if(cost < min_cost) {
            min_cost = cost
            selectBestTraj(trajectory)
        }
    }
}
```

Figure 2.3: Pseudo code for trajectory generation

There is also computation for each point in each trajectory, which can be seen in the pseudocode described in Figure 2.3.

2.1.3 Cost function

To determine the best trajectory, a cost function is applied to each generated trajectory. The cost function is defined as achieving human-like driving trajectories. In this case, ξ represents the vector of a target state, either longitudinal or lateral. Different weights k represent different penalisations in the cost function, namely slow convergence, final deviations from reference path and prioritising longitudinal or lateral movement. Final deviation and movement are defined by positional offset and convergence in jerk, which is the derivative of acceleration.

$$J[d,s] = J_d[d] + k_s J_s[s], \quad k_s > 0 \tag{2.2}$$

$$J_\xi := f_0(u)dt + (h(\xi(t),t))_\tau \tag{2.3}$$

$$f_0(u) := \frac{1}{2}u^2(t) \tag{2.4}$$

$$h(\xi(t),t) := k_\tau t + \frac{1}{2}k_{\xi 1}[\xi_1(t) - \xi_r ef(t)]^2, \quad k_\tau, k_{\xi 1} > 0 \tag{2.5}$$

2.1.4 Object collision and curvature checking

Apart from the cost functions described in Equation 2.2, each trajectory has to be checked for collisions with obstacles and maximum turning curvature. Naturally, a

trajectory should be discouraged if it risks colliding with an obstacle, which can be done by increasing its cost by a considerable amount. Furthermore, for the vehicle to execute a trajectory, it has to be within the maximum curvature range to work.

2.2 Parallel architectures & optimisation strategies

A fundamental understanding of multicore architectures for CPU and GPU is required to understand why certain optimisation strategies work. A parallel programming API for CPU and GPU is presented to understand implementation details.

2.2.1 GPU architecture

This section explains how a GPU's architecture is designed, especially regarding parallel computing. Since the architectural terminology differs between vendors, equivalent technologies can be seen in Table 2.1. The target GPU of the thesis is manufactured by Nvidia, and thus, the paper will use Nvidia terminology.

NVIDIA	**AMD**	**Description**
Thread	Work-item	Unit of execution
Warp	Wavefront	Group of threads executed with SIMD
Block	Workgroup	Collection of warps/wavefronts
Grid	NDRange	Collection of blocks/workgroups
SM	CU	Processing units

Table 2.1: GPU terminology mapping between Nvidia and AMD

2.2.1.1 Grid, block & warp

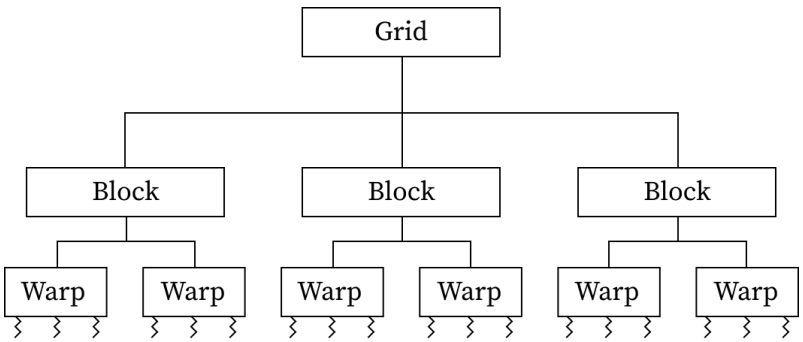

Figure 2.4: GPU thread grouping architecture

In CUDA terminology, warp is the name for a small subset of threads that execute the same instruction in parallel. This action of executing a single instruction on multiple data is a computation model often abbreviated to *SIMD*. In the context of GPUs, SIMD is usually called *SIMT* instead since the instruction is running on multiple threads. GPUs are a suitable hardware target for the SIMD computational

model due to the low scheduling overhead compared to a CPU [5]. A warp size of 32 threads is used for all current Nvidia GPU architectures. An instruction is loaded to all threads within a warp, which is then executed on all threads simultaneously, and only after all threads have finished their execution the next instruction is loaded [5].

A group of warps is called a *block*. Blocks are used to schedule the execution of warps onto different execution units on the GPU. They also provide isolation from blocks of other warps, meaning that they can execute in any order. Threads within a block are allowed to have independent synchronisation within itself. A block also provides a local shared memory that can be used by all threads within [5]. A *grid* is the highest thread organisation level containing blocks. A grid represents the total parallel computation for a GPU kernel and determines the number of blocks to execute. A grid can be defined as a 1D, 2D or 3D grid of blocks [5]. Figure 2.4 illustrates the hierarchical relation between the concepts.

2.2.1.2 CUDA

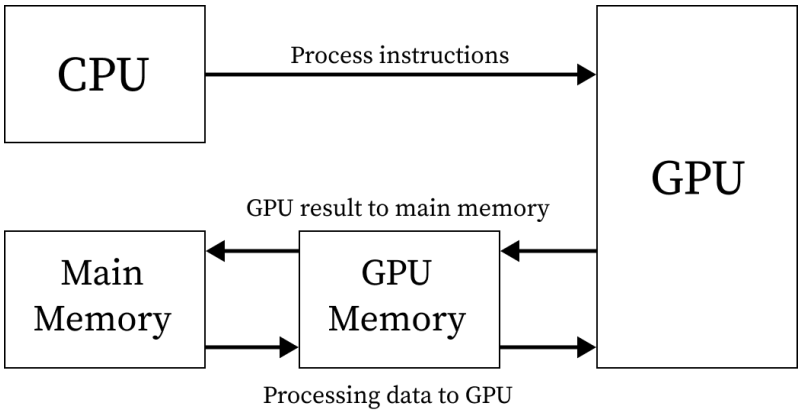

Figure 2.5: CUDA processing flow

CUDA is used to access the GPU's virtual instruction set in all instances of GPU-acceleration. The software layer is a proprietary API developed by NVIDIA for a specific selection of their GPUs. CUDA focuses on exposing elements for parallel computation to ease compute kernel acceleration. CUDA is designed for different languages, whereas the C/C++ compiler is *nvcc*. The processing flow of CUDA can be seen in Figure 2.5. The CPU supplies the GPU with kernel instructions and memory portions to copy. GPU kernels are executed in parallel and copy the result to the main memory [5].

CUDA functions are compiled into Parallel Thread Execution (PTX) assembler code for the GPU. The CUDA kernel initiates grid generation, which schedules and synchronises the parallel thread execution within its blocks and warps. The CPU allocates memory in the GPU and copies the processing memory into the GPU memory. Processing instructions in the form of PTX assembler instructions are also sent to the GPU. When execution is complete, the result is copied back into the main memory, and the CPU can free up its previous allocation of GPU memory.

2.2.1.3 Streaming Multiprocessors

NVIDIA GPUs consist of execution units called Streaming Multiprocessors (SMs), which, in turn, consist of several execution units called streaming processors (SPs) that can execute one instruction at a time for a specific thread [6]. To schedule a thread block for execution, it must be mapped onto an SM. Each SM can have several blocks mapped to it. However, it can only execute one warp in a block at a time. If an executing warp encounters a long latency operation, the SM can context switch to a different warp in the block to hide the latency of that operation. Therefore, to gain high SM utilisation, it is preferable to organise the threads into blocks of adequate sizes so that the SM can perform latency hiding for long latency operations.

2.2.2 CPU multicore architecture

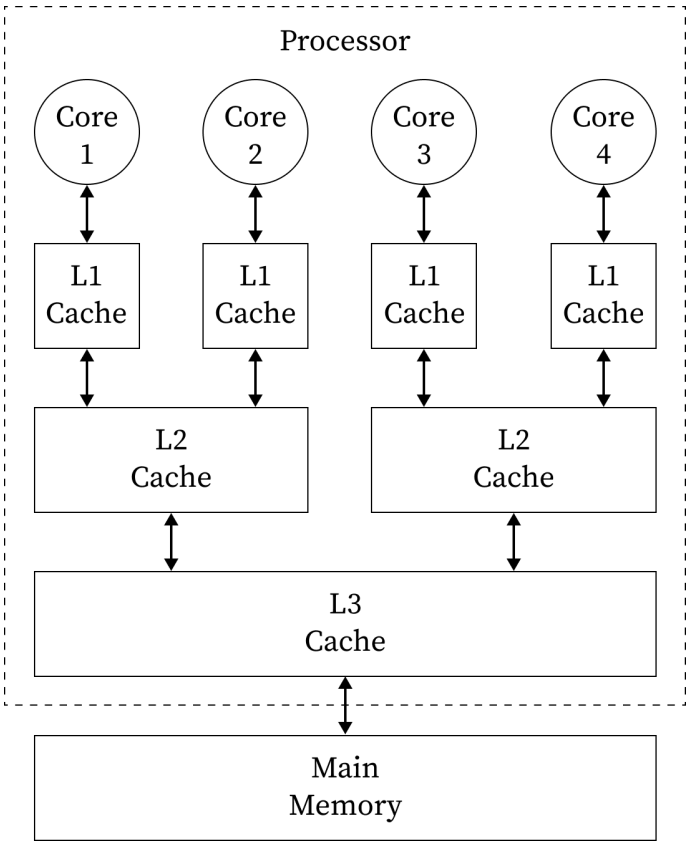

Figure 2.6: Example architecture of a multicore processor

Each core in Figure 2.6 is responsible for executing program instructions. The cores are connected to cache memories, which hold data used in computations. The layout in Figure 2.6 is an example of core-cache relation; specific models may differ.

2.2.2.1 OpenMP

OpenMP is an API that provides directives for multiple data constructs on a multicore CPU [5]. It provides a way to define private data, shared data, and several threads

to run in a parallel region. The library targets several languages, most importantly C++, for the analysed multicore implementation. CPU threads are created and then destroyed in a fork-join model, meaning all threads must finish before the parallel region is deemed executed, and the next part can continue.

The OpenMP API offers a multitude of preprocessor directives or `pragmas` that the programmer can use to parallelise code regions. These pragmas are applied to code blocks as shown in the example in Figure 2.7. Depending on the directive, one or more clauses can be added to modify its behaviour. For example, to parallelise the iterations of a for loop such that a pool of worker threads each collaborate to finish the for loop, the directive `parallel for` can be added to the for loop as shown in Figure 2.8. In this example, the clauses `num_threads` and `schedule` are also applied, changing the number of threads in the worker pool and the scheduling algorithm determining how iterations are distributed among worker threads. At the end of the parallel region, an implicit barrier ensures that each thread has finished executing the parallel region before continuing program execution beyond the region [5].

```
#pragma omp directive [clauses]
{ block }
```

Figure 2.7: Example of OpenMP pragma applied to a code block.

```
#pragma omp parallel for num_threads(4) schedule(dynamic)
for (int i = 0; i < num_iters; i++) {
    ...
}
```

Figure 2.8: Example of OpenMP pragma applied to a code block.

OpenMP also provides synchronisation directives that can be used within a parallel region to ensure correctness. For example, the `atomic` directive can be applied to a code line to ensure the instruction is atomically performed. Furthermore, `critical` can be used to define a critical region within a parallel region, and `barrier` can explicitly specify a barrier in a parallel region [5].

2.2.3 Memory locality

The memory hierarchy of computer systems allows for performance optimisations in kernels with a strong locality of reference. A non-exhaustive representation of memory levels relevant to this thesis can be seen in Figure 2.9. When a specific memory location is accessed, the memory is loaded into a cache block in the CPU memory. Every access that is another memory address will start demoting the cached memories level in the hierarchy. First, it will demote into lower cache levels and, lastly, into the main memory.

This hierarchy also represents read/write speed when communicating with the CPU. The highest cache level will have the shortest access time while main memory will

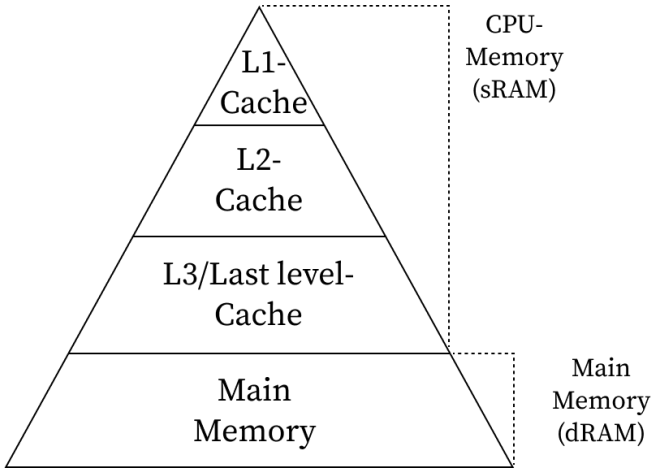

Figure 2.9: Non-exhaustive memory hierarchy

have the longest [5]. This thesis primarily focuses on two types of locality: temporal and spatial locality. Temporal locality is when a specific cache block is accessed several times before being replaced. Thus, high temporal locality will keep the memory in a lower cache level and minimise read/write time to the memory location [5]. Spatial locality describes how programs often access memory locations with neighbouring addresses. Therefore, in a program that utilises spatial locality, memory access will be followed by successive access in the same cache. Thus, the program can utilise a lower-level cache's short read/write times for multiple memory locations [5].

2.2.4 Memory access patterns & Data layout

Since levels of memory locality lead to shorter execution times through reduced read and write times, the memory access pattern is essential. For example, accessing memory through nested pointers to different locations in memory results in bad memory locality, as cache lines are less likely to be reused. However, if the data is structured and accessed in a way that provides good memory locality, performance can significantly increase [5].

Two data modelling alternatives that are commonly discussed are Structure of Arrays (SoA) and Array of Structures (AoS). Examples of these data layouts are showcased in Figure 2.10. Both layouts in this figure represent the same thing: a collection of values of types A, B and C. However, their structure can significantly affect the application's performance, depending on the access pattern [7]. Furthermore, it can be said that the SoA layout is preferred when writing parallelisable code, as it is easier for C++ compilers to vectorise [8].

Depending on the kernel, data layout and memory access pattern are essential when writing high-performance GPU applications. Also, with CPUs, GPUs have a memory hierarchy, where the L1 cache for each SM is the fastest, then a shared L2 cache, and the global memory, which is the slowest [9]. When a warp thread accesses global memory, depending on the access pattern, the accesses can be coalesced into a single

```
// Structure of Arrays
struct N_ABCs {
    TypeA as[N];
    TypeB bs[N];
    TypeC cs[N];
};

// Array of Structures
struct ABC {
    TypeA a;
    TypeB b;
    TypeC c;
};
ABC abcs[N];
```

Figure 2.10: Structure of Arrays (SoA) and Array of Structures (AoS) data layout examples in C++.

memory access instead of one for each thread in the warp. The coalescing is only possible if the threads within the warp access contiguous memory locations [9], which the SoA data allows. This way, the memory throughput is maximised, and the application execution time is decreased because of less memory latency.

3

Implementation

This chapter provides a detailed description of the steps taken to implement the three versions of the algorithm investigated by the thesis. Firstly, the sequential implementation of the algorithm will be described and used as a base for implementing the multicore and GPU-accelerated versions of the algorithm. Subsequently, implementations of the multicore and GPU-accelerated versions are described. Intermediary results are also presented throughout the chapter to motivate implementation decisions before reaching a final implementation. Verification tool implementations are also given to ensure algorithmic correctness between implementations.

3.1 Verification

During verification, a way must be found to consistently generate plausible scenarios and verify that the picked trajectory is correct. Instead of coupling the application to closed source libraries, a custom testing framework was developed.

3.1.1 Deterministic property-based testing

The framework is loosely coupled to the algorithm and is designed around property-based testing. The input generation uses a pseudo-random number generator to create realistic traffic scenarios and roads. The result of the random number generator is deterministic based on a *seed*, enabling debugging and verification for specific scenarios while still enabling testing an endless amount of scenarios. The vehicle state is generated based on bounded random number generation, however, the road is generated by creating a cubic spline between pseudo random points.

3.1.2 Manual visualisation

To verify the correctness of the implementation, Python programs parse the output data of the algorithm and display a visualisation of the trajectories. The program also displays the longitudinal and lateral components and their first, second and third derivative of each trajectory in the frenet frame. Manual visualisation is an essential step in ensuring correctness during optimisations. The coupling is implemented by generating output JSON data, which is enabled via preprocessor directives, removing JSON processing when benchmarking. The Python programs parsed the output to JSON, which displays the data as a collection of interactive graphs, as seen in

Figure 3.1. Trajectories are color-mapped, where red defines low costs, and blue has high costs. Red circles shown only in the top graph are obstacles associated with their possible trajectories. The green line is the reference path, and the line with increased width is the trajectory with the lowest cost. The costs of the trajectories were also printed by the program for debugging reasons. Introduced bugs could then be identified if there was any difference in the trajectory costs between versions.

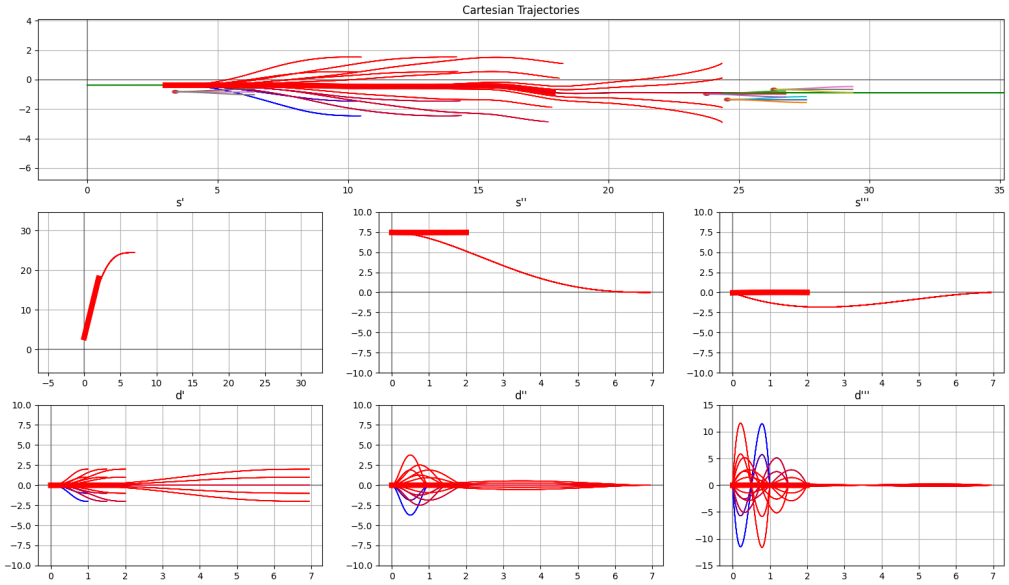

Figure 3.1: Example of visualisation tool output. Top graph represent trajectories and obstacles generated. Middle graphs represents first, second and third derivative of the longitude. Bottom graphs represent first, second and third derivate of the latitude.

3.2 Sequential implementation

The sequential implementation of the algorithm was developed in C++, with the help of visualisation tools for the algorithm output data written in Python. The Python tools assisted the implementation by visualising the results and confirming correctness. Data types in the implementations use single precision floats.

3.2.1 Initial state generation

Executing the trajectory planning algorithm requires data about the current state. The state includes information about the car, detected obstacles and reference path. The car's current state includes its heading, position, velocity and acceleration. Each detected obstacle has a position, radius, and three paths it might traverse, which are referred to as obstacle trajectories. Lastly, the reference path is defined as the centre of the lane or road being traversed. This data is generated using a mock state generator that produces a pseudo-random state depending on its seed. The attributes of the generated state are bounded so that completely unrealistic scenarios

do not occur. For example, the initial velocity of the car is randomised within a certain bound that represents realistic driving speeds.

3.2.2 Trajectory generation

When a mock state has been generated, it is fed to the initial algorithm function responsible for developing and evaluating trajectories. Initially, the lateral and longitudinal driving strategies are iterated. The lateral strategies are lane keeping and lane changing, and the longitudinal strategies are velocity keeping and stopping. Next, the driving strategy and mock state are fed to a function computing a list of target times. The current strategy and target times are then used to calculate the lateral and longitudinal target states representing the car's lateral and longitudinal position, velocity and acceleration at the specified target time.

Thereafter, the algorithm iterates over longitudinal and lateral offset ranges to generate more trajectories that deviate from the target state. Then, a quintic polynomial representing the lateral trajectory movement is generated, along with a quintic or quartic polynomial for the longitudinal component, depending on the driving mode. These polynomials' values, along with their first to third derivative, are then sampled with a configurable resolution to generate a vector of trajectory points in the frenet frame. The polynomials are only sampled up to the current target time being iterated over.

As the points are being sampled, they are also iteratively converted into the cartesian coordinate system. The conversion is done by finding the point on the reference path representing the same traversed longitudinal distance as the longitudinal frenet trajectory sample. To find this point, the algorithm integrates the distance from the current reference point until it is greater or equal to the longitudinal sample value. The point is then used to convert the frenet sample into cartesian space. The frenet sample point and its cartesian representation are saved in vectors that are then used to calculate the cost of the trajectory according to the functions described in Equations 2.2 to 2.5. Pseudocode for the trajectory generation function can be seen in A.1.

3.2.3 Collision detection and curvature checking

The cost of each trajectory is not only affected by the formulas in 2.2, it is also affected by a collision checking routine that penalises trajectories on collision paths with obstacles. Each obstacle in the algorithm has a radius, position, and several possible trajectories that it might traverse along with some probability. To determine if a trajectory is on a collision path with an obstacle's trajectory, each pair of points in both trajectories has to be farther apart than the radius of the obstacle.

The pseudocode displayed in Figure 3.2 shows how the points of each obstacle trajectory are checked against each point in a generated trajectory to determine if there is a collision. This process is repeated for each of the generated trajectories, and the cost of trajectories with possible collisions is modified. The added cost for a collision-trajectory is calculated as in Equation 3.1, where $K_{collision}$ is some constant

```
for (traj in trajectories) {
    for (obs in obstacles) {
        for (obs_traj in obs.trajectories) {
            for (p1 in obs_traj) {
                for (p2 in traj) {
                    if (distance(p1, p2) < obstacle_radius) {
                        traj.cost += coll_cost;
                        goto next_obs_traj;
                    }
                }
            }
        }
    next_obs_traj:;
    }
}
```

Figure 3.2: Pseudo code for collision detection.

representing the unweighted cost of a collision. This value is then added to the computed cost value from Equation 2.2.

$$cost_{collision} = K_{collision} * P_{obs\_traj} \tag{3.1}$$

The algorithm presented in [1] does not describe this collision detection strategy. However, this strategy was chosen to better account for the unpredictability of dynamic obstacles in a real-world scenario.

Furthermore, since trajectories have different target times and, therefore also, various lengths, it is essential to compare their costs fairly. In Werling et al. [1], the strategy implemented is to truncate longer trajectories and extrapolate shorter trajectories to evaluate the costs of the trajectories at the same time. However, the approach used in this implementation is to divide the total cost by the number of points evaluated on the trajectory, which is proportional to its target time. This strategy avoids extrapolating the trajectory polynomials beyond their defined range.

3.3 Parallelisation analysis

Before implementing a multicore and GPU-accelerated version of the algorithm, a thorough parallelisation analysis was conducted to identify bottlenecks and parallelisation opportunities in the sequential implementation. This section comprehensively describes these opportunities and how they were identified using performance analysis tools.

3.3.1 Collision detection

The first step of the parallelisation analysis was generating a flame graph of the sequential execution. A filtered view of this flame graph showcasing the algorithm execution is displayed in Figure 3.3. This graph shows that most of the execution time is spent checking the trajectories for collisions, which indicates that effort should be put into optimising this step.

Figure 3.3: Time spent in functions of the sequential execution. The view is filtered only to show the algorithm execution.

When analysing the collision checking kernel, it is apparent that the outer for-loop is entirely parallelisable since there are no inter-loop dependencies. Also, parallelising any other loop would require synchronisation or atomicity when adding the trajectory cost. Furthermore, flattening the three outer loops could result in better load balancing since trajectories differ in the number of points sampled, proportional to their target time. Parallelisation of this kernel could lead to significant improvements such that the trajectory generation accounts for a substantial portion of the total execution time.

3.3.2 Trajectory generation

Parallelisation of this kernel proved more challenging than the collision-checking kernel. As seen in the pseudocode in Figure A.1, the outer loop is entirely parallelisable as there are no inter-loop dependencies. However, as there are only two driving strategies in each outer for loop, parallelisation of each iteration would result in only two threads running in parallel, insufficiently utilising the available hardware. Under-utilisation is the case for all loops except the innermost loop. The innermost loop has loop-carried dependencies through the `sum` variable that is used to significantly reduce the execution time of converting the frenet trajectory point to cartesian coordinates. Therefore, it is not parallelisable.

An alternative parallelisation strategy would be to coalesce all but the inner for loop so that each thread generates a trajectory and evaluates it. The approach would result in more fine-grained parallelism. Each thread would be assigned a longitudinal and lateral strategy, a target time, and lateral and longitudinal offsets in this strategy. This combination of parameters can be precomputed such that a list of all parameter combinations can be iterated in a parallel manner, as each iteration would be independent. For an example showcasing how this combination list can be created, see Figure 3.4.

When trajectory generation and sampling have finished, the trajectory is saved in a dynamic vector as shown in Figure A.1. It's important to note that this has to

```
Combinations all_combs;

for (auto lat : lateral_strategies) {
    for (auto lon : longitudinal_strategies) {
        const auto frenet = cartesianToFrenet(ego_data, ref_data);
        auto target_times = getTargetTimes(frenet, lon_strat);

        for (auto target_time : target_times) {
            for (auto d : d_steps) {
                for (auto s : s_steps) {
                    all_combs.append({lat, lon, target_time, d, s});
                }
            }
        }
    }
}
```

Figure 3.4: Pseudocode of trajectory generation combinations list creation.

be done thread-safe using some synchronisation mechanism if the underlying data structure is not inherently thread-safe. Thread safety is not considered in standard library C++ vectors and, therefore, warrants using a synchronisation mechanism like a semaphore to avoid race conditions. However, if trajectories are stored in an array where each trajectory has a pre-assigned index, this would not be a problem. The main drawback of this approach is the overhead incurred by generating the list of combinations.

3.4 Multicore implementation

In the multicore implementation, the collision checking kernel and the trajectory generation kernel were parallelised using OpenMP. The statistics presented in this section were gathered by benchmarking the algorithm with 50 different seeds. Furthermore, the same discretisation parameters were used in each execution, which are listed in Table 3.1. The number of obstacles and the obstacle trajectory parameters are also listed as factors that significantly affect the performance of collision checking.

3.4.1 Collision checking

Initially, the collision checking kernel was parallelised using the OpenMP pragma `#pragma omp parallel for scheduling(x) num_threads(y)` where x is the scheduling strategy, and y is the thread count. This pragma was placed on the outer for loop, iterating over trajectories. Thread counts 4, 8 and 16 were considered, along with static, dynamic and guided scheduling strategies. Results of these benchmarks are shown in Table 3.2 and visualised in Figure 3.6. Dynamic scheduling with different block sizes was also considered but did not yield any performance increase and was excluded from the table and graph.

Table 3.1: Benchmarking algorithm parameters used.

Parameter name	Count	Note
Lateral strategies	2	Lane keeping, Lane change
Longitudinal strategies	2	Velocity keeping, Stopping
Target times	3	Depends on strategy
Lateral offsets	5	-2m to 2m, 1m steps
Longitudinal offsets	3	-0.1m to 0.1m, 0.1m steps
Total trajectory count	180	
Obstacles	10	Amount of obstacles
Obstacle trajectories	3	Trajectories per obstacle
Obs. trajectory length	200	Points per obs. trajectory

Table 3.2: Benchmark results of the unflattened collision check kernel. Execution times are measured in microseconds.

Thread count	static (µs)	dynamic (µs)	guided (µs)
4	151297.95	100782.92	97400.25
8	80047.17	53326.32	50219.68
16	69597.14	54168.8	54035.19

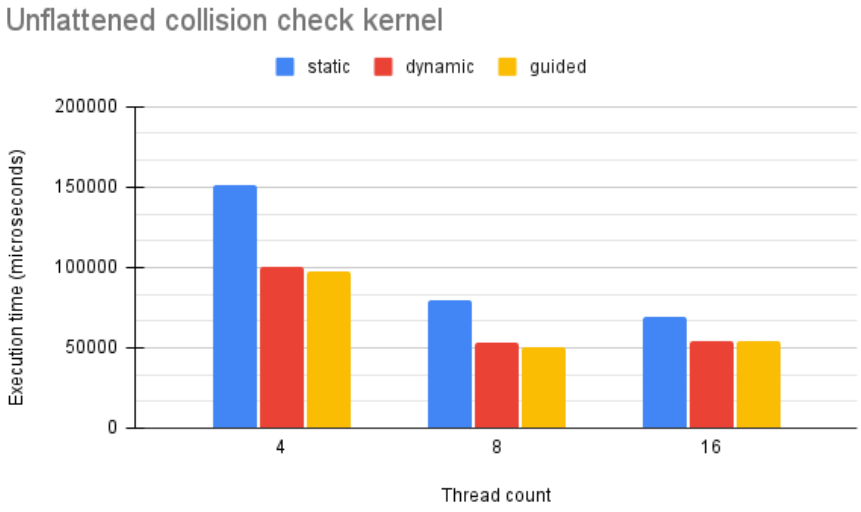

Figure 3.5: Benchmark results of the unflattened collision check kernel.

Table 3.2 shows that guided scheduling with eight threads is optimal. The impact of coalescing the kernel was also benchmarked, and these results are shown in Table 3.3 and Figure 3.5.

Table 3.3: Benchmark results of the flattened collision check kernel.

Thread count	static (µs)	dynamic (µs)	guided (µs)
4	151549.63	105631.4	97074.09
8	75567.38	57075.6	49994.42
16	75402.51	53146.58	49858.04

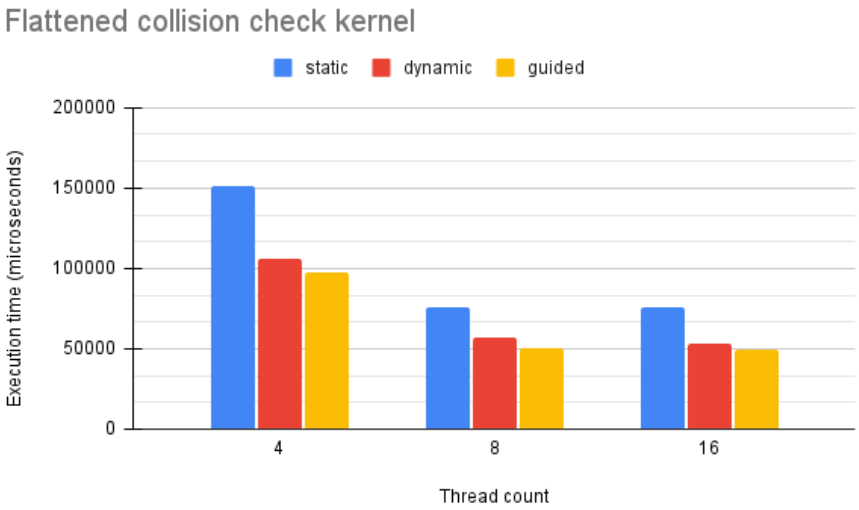

Figure 3.6: Benchmark results of the flattened collision check kernel.

As can be seen in Tables 3.3 and 3.2, there was no significant performance increase of the flattened kernel. However, the flattened kernel is marginally faster with 16 threads and guided scheduling and was kept. Also note that the dynamic scheduling strategies are significantly better than static scheduling. This is because of load imbalance implied by some trajectories having more points to check than others. There is also an early return statement in the collision checking code that proceeds to the next obstacle trajectory if a collision is found as can be seen in Listing 3.2. This way, trajectories that collide early will finish collision checking faster. By dynamically being able to reassign iterations to different threads, the scheduler can account for this load imbalance.

3.4.2 Trajectory generation

After optimising the collision checking kernel, it became apparent that trajectory generation consumed a significant portion of the total execution time, as seen in Figure 3.7. This data was collected using PAPI as flame graphs become unintuitive visualisations in multicore environments since the stack traces include significant thread management noise. As stated in Section 3.3.2, to parallelise the trajectory

generation kernel efficiently, one should flatten the outer loops and parallelise the iteration over parameter configurations much like the flattened collision checking kernel. The flattening was implemented using a pre-allocated array where each trajectory has an index to avoid synchronisation mechanisms.

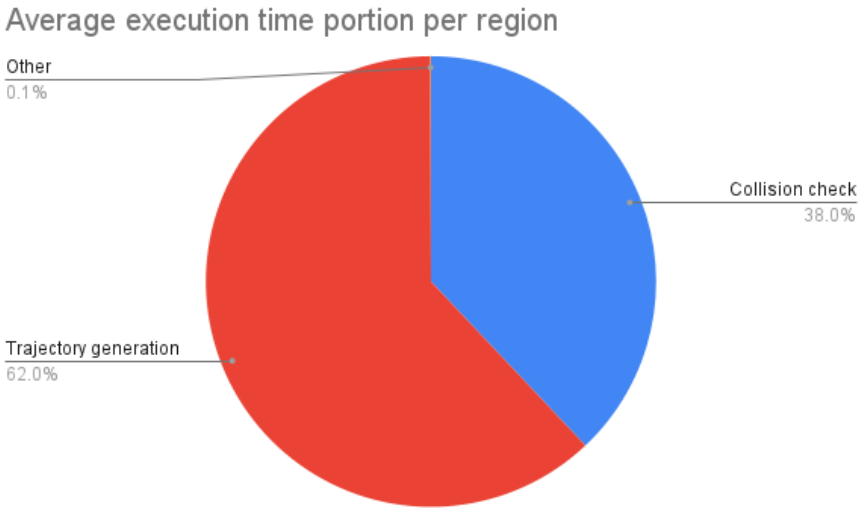

Figure 3.7: Average portion of execution time of each region after collision check optimisation.

As with the collision checking kernel, the flattened trajectory generation kernel was parallelised using an OpenMP pragma on the outer loop. The thread counts 4, 8 and 16 were considered, as well as static, dynamic and guided scheduling strategies. The results are shown in Table 3.4 and Figure 3.8.

Table 3.4: Benchmark results of the flattened trajectory generation kernel.

Thread count	static (µs)	dynamic (µs)	guided (µs)
4	30781.89	22964.16	21564.23
8	20085.06	11663.17	11698.41
16	10417.27	6772.51	6692.32

An unflattened version of the trajectory generation kernel was also parallelised using an OpenMP pragma on the outer loop. However, this did not yield good results, as the iterations had to be more numerous to efficiently use the available cores. Consequently, these performance results are not shown in this section.

3.5 GPU-accelerated implementation

To GPU-accelerate the algorithm, a series of steps had to be taken. First, a number of refactors had to be done to achieve CUDA compatibility. Then, each kernel was rewritten into GPU-kernels one by one, iteratively exploring performance metrics to find the best approach. Finally, efforts were made to keep as much computation

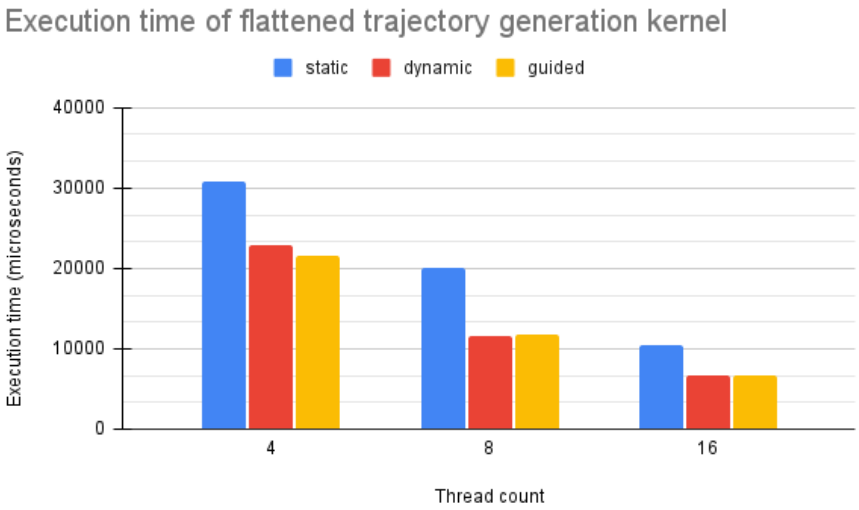

Figure 3.8: Benchmark results of the unflattened collision check kernel.

on the device as possible to avoid unnecessary memory transfers between CPU and GPU.

3.5.1 CUDA-compatibility refactor

To transfer memory and execute kernels in a CUDA context, some data structures had to be significantly refactored to avoid non-compatible structures like C++ vectors, replacing them with statically sized arrays of structures instead. Dynamically sized CUDA-compatible data structures are available through the `thrust` library included with the CUDA toolkit. This option made the refactoring process less labour-intensive and avoided potential memory leakage problems. However, due to the memory management overhead required when using dynamic memory structures, it was decided that statically sized arrays would be used instead. This refactor was back-ported to previous implementations to keep the comparison fair.

3.5.2 Collision check GPU implementation

When designing the kernel, a granularity would have to be decided. The multicore implementation had a relatively large granularity as it operated per trajectory. For the case where 180 trajectories are generated, it would not make efficient use of all available GPU threads. The collision check loop-nesting was flattened to achieve a more suitable granularity for the GPU kernel, as seen in Listing 3.9. After being flattened, each thread executes one position of one obstacle trajectory against the entire path of one trajectory. The flattening resulted in a much more fine-grained granularity, which is a better fit for the GPU. The second strategy was to change from AoS to SoA for the trajectories and obstacles of data structures. The data structures were also completely flattened, as seen in Listing 3.9. Furthermore, the flattening not only significantly improved the time to copy data to the device but also lowered the execution time of the kernel by a factor of roughly 20. Flattening also made most of

```
const int th_idx = blockIdx.x * blockDim.x + threadIdx.x;
if(th_idx >= THREAD_COUNT) return;

const int obs_traj_pos_idx = th_idx % OBS_PATH_LEN;
int temp = th_idx / OBS_PATH_LEN;

const int obs_traj_idx = temp % OBS_TRAJ_COUNT;
temp /= OBS_TRAJ_COUNT;

const int obs_idx = temp % OBS_COUNT;
const int traj_idx = temp / OBS_COUNT;

const int start_idx = traj_start_indices[traj_idx];
const int end_idx = traj_start_indices[traj_idx+1];

const float radius = obs_radii[obs_idx];
const float radius_sqrd = radius * radius;

const int flat_pos_idx = obs_idx * (OBS_TRAJ_COUNT * OBS_PATH_LEN)
+ obs_traj_idx * OBS_PATH_LEN + obs_traj_pos_idx;
const float obs_traj_x = obs_x_arr[flat_pos_idx];
const float obs_traj_y = obs_y_arr[flat_pos_idx];

for(int idx = start_idx; idx < end_idx; idx++) {
    const auto dx {obs_traj_x - traj_x_arr[idx]};
    const auto dy {obs_traj_y - traj_y_arr[idx]};
    const auto dist {dx*dx + dy*dy};

    if(dist < radius_sqrd) {
        const int probs_idx = obs_idx * OBS_TRAJ_COUNT + obs_traj_idx;
        atomicAdd(&costs[traj_idx], KCOLL * obs_probs[probs_idx]);
        break;
    }
}
```

Figure 3.9: Pseudocode of flattened GPU kernel

the GPU activity memory transfers instead of computation, which made streaming the data to the GPU viable. This optimisation was also backported to previous versions. Streaming further reduced the execution time, allowing computation to be overlapped with data transfers.

Attempts were made to further increase the granularity of the kernel by flattening the inner for-loop seen in Figure 3.9. However, this resulted in a granularity far too high to be feasible, as it resulted in a tenfold increase in the execution time of the kernel. Furthermore, the memory access pattern to the global memory of this kernel is not optimal, as the memory accesses are not coalesced. The only repeated necessary global memory access is to the `d_y` and `d_x` arrays. These accesses need to be coalesced or cached in shared memory to reduce the latency inferred by these accesses. However, shared memory is only feasible if one thread block handles collision checking for an entire trajectory, which is not a desired approach as the granularity becomes too coarse. Furthermore, the memory accesses cannot be coalesced as one thread accesses all of the memory locations in the trajectory.

3.5.3 Trajectory generation

The first approach was to run a task per trajectory generated. This granularity was too coarse to run efficiently on a GPU and yielded no significant speedup compared to the sequential version. The second approach separated the logic of sampling the trajectory and the creation of the quintic and quartic polynomials. The polynomial creation isolated the complex parts of the generation and was kept on the CPU while the trajectory sampling could be moved to the GPU. Depending on the length of the trajectories and path resolution, the granularity changes, but a typical run spawns about 50000 tasks, one for each position to sample. The reduced granularity resulted in a significant speedup, about a tenfold improvement in runtime from the sequential implementation.

The vast majority of the execution time in the trajectory sampling kernel now went towards finding the time travelled at a certain arc length of the reference path needed to convert the sampled frenet frame points back into the cartesian coordinate system. The accumulated arc traversal was precomputed with a 0.05 resolution on the CPU to make this kernel more suitable for a GPU. This precomputed data could then be traversed with a binary search on the GPU in runtime. The pre-computation, along with the binary search, decreased the overall execution time by a factor of approximately 5.

Figure 3.10 displays the pseudocode of the trajectory sampling kernel. The thread calculates which trajectory point to sample using its index `p_idx`, and precomputed arrays `traj_indices`, `start_indices` that are designed for this purpose. The array `traj_indices` helps the GPU thread find the trajectory index on which it is supposed to sample a point. The `start_indices` includes information about which index the trajectory to be sampled is located in the flattened trajectory data structure. The `start_indices` allows the GPU thread to compute which time `t` the thread should sample the trajectory at. When the frenet data has been sampled and stored into the `d_s` and `d_d` arrays, the point is converted back into the cartesian coordinate

system, which is also stored in device memory.

```
int p_idx = blockIdx.x * blockDim.x + threadIdx.x;
int traj_idx = traj_indices[p_idx];
int start_idx = start_indices[p_idx];
int offset_from_start = p_idx - start_idx;

float t = SAMPLE_STEP * offset_from_start;
QuinticPolynomial sqp = sqps[traj_idx];
QuinticPolynomial dqp = dqps[traj_idx];

mem->d_s1[p_idx] = at(sqp, t);
mem->d_s2[p_idx] = firstPrime(sqp, t);
mem->d_s3[p_idx] = secondPrime(sqp, t);
mem->d_s4[p_idx] = thirdPrime(sqp, t);

mem->d_d1[p_idx] = at(dqp, t);
mem->d_d2[p_idx] = firstPrime(dqp, t);
mem->d_d3[p_idx] = secondPrime(dqp, t);
mem->d_d4[p_idx] = thirdPrime(dqp, t);

auto ref_time = mem->d_csp->findTimeAtLength(mem->d_s1[p_idx]);

tpa::ReferenceData target_ref = {
    mem->d_csp->calcX(ref_time),
    mem->d_csp->calcY(ref_time),
    ref_time,
    mem->d_csp->calcOrientation(ref_time),
    mem->d_csp->calcCurvature(ref_time),
    mem->d_csp->calcCurvatureRate(ref_time),
};

frenetToCartesian(mem, p_idx, &target_ref);
```

Figure 3.10: Pseudo code of trajectory sampling kernel

Using shared memory in the trajectory sampling kernel is not beneficial, as there is no data reuse or cooperation between threads. Each thread performs several reads and writes to global memory, but other threads cannot reuse the data read in the same block.

3.6 Keeping computation on device

Memory transfers to and from the device can be time-consuming, so reducing the amount of memory transfers conducted by the program is beneficial. To this end, the algorithm was modified to eliminate unnecessary memory transfers to and from the device between the execution of GPU kernels. Ideally, no data should have to be

copied back to the CPU except the lowest cost trajectory. To enable this, the cost functions were also converted to GPU kernels.

3.6.1 Cost functions

To convert the cost computation into GPU kernels, it was split into two separate kernels responsible for computation and accumulation. In the cost computation kernel, each GPU thread computes the cost contribution of one sampled point along a trajectory. This kernel has the same granularity as the trajectory sampling kernel. The second kernel is then used to accumulate the costs computed by the first kernel, one GPU thread per trajectory. The granularity of the second kernel is suboptimal for GPU execution. Still, this strategy eliminates the need to copy all trajectory data back to the host to perform the computation there, which would result in significant memory transfer execution times. Furthermore, the execution time of the second kernel is so insignificant that the granularity does not matter.

The first kernel functions by each thread atomically adding the jerk cost to a cost array with one element for each trajectory. The atomic operation is critical since all the threads operating on the same trajectory must add their computed cost to the total trajectory cost. The same is true for curvature violations, but an atomic OR operation is performed instead as it is a boolean value. The second kernel reads these values for each trajectory to compute the final cost.

The addition of these two cost kernels allowed the entire algorithm, from trajectory sampling to finding the winning trajectory, to be executed on the GPU without transferring memory to and from the host in the intermediate steps. Only the cost array is copied back to the host to find the winning trajectory, where the minimum cost and its corresponding trajectory index are found. That index is then used to copy back the data of the winning trajectory to the host, which would then be executed by the car in a real scenario. Only copying back the cost and the winning trajectory resulted in another significant reduction of execution time, as device-to-host data transfer was a substantial portion of the execution time.

The kernel call procedure of the algorithm is shown in Figure 3.11. First, the trajectory sampling kernel is called with a preconfigured block size. The grid size is calculated from the block size so that all points for each trajectory are sampled. After the sampling kernel is called, `cudaDeviceSynchronize` is called to ensure that all trajectories have finished sampling before evaluating the trajectories. The same is true after the `totalCostFineGrained` kernel call, and each cost calculation needs to be completed before accumulating the values. Note that each kernel call includes `mem.stream`. The synchronisation is necessary because each memory transfer is streamed to enable data-computation overlapping to reduce execution time further. After the collision checking kernel, a call to `cudaStreamSynchronize` is performed, ensuring that each memory transaction using the stream is finished before transferring memory back to the host.

```
constexpr auto traj_sampl_block_size {TRAJ_SAMPLING_BLOCK_SIZE};
const auto traj_sampl_grid_size =
{((coords_len) + traj_sampl_block_size - 1) / traj_sampl_block_size};
trajectorySampling<<<traj_sampl_grid_size,
traj_sampl_block_size,
0,
mem.stream>>>(d_mem, d_dqps, d_sqps, d_target_times);
cudaDeviceSynchronize();

constexpr auto total_cost_block_size {TOTAL_COST_BLOCK_SIZE};
const auto total_cost_grid_size =
{((coords_len) + total_cost_block_size - 1) / total_cost_block_size};
totalCostFineGrained<<<total_cost_grid_size,
total_cost_block_size,
0,
mem.stream>>>(d_mem, d_target_times);
cudaDeviceSynchronize();

const auto accumulate_block_size {trajectory_count};
constexpr auto accumulate_grid_size {1};
accumulateCosts<<<accumulate_grid_size,
accumulate_block_size,
0,
mem.stream>>>(d_mem, d_d_targets, d_s_targets, d_target_times);

constexpr auto coll_check_block_size {COLLISION_CHECK_BLOCK_SIZE};
const auto coll_check_grid_size
{(mem.thread_count + coll_check_block_size - 1) / coll_check_block_size};
collisionCheck<<<coll_check_grid_size,
coll_check_block_size,
0,
mem.stream>>>(d_mem);
cudaStreamSynchronize(mem.stream);
```

Figure 3.11: Kernel calls in the GPU implementation of the algorithm.

The final GPU-accelerated version achieved execution times shown in Table 3.5 with the discretisation parameters shown in Table 3.1. The proportions are visualised as a pie chart in Figure 3.12.

Table 3.5: Execution time total and per region for GPU implementation. Benchmarking parameters shown in 3.1

Region	Median execution time (μs)
Total	2613.24
Collision check	1324.94
Trajectory generation	998.48

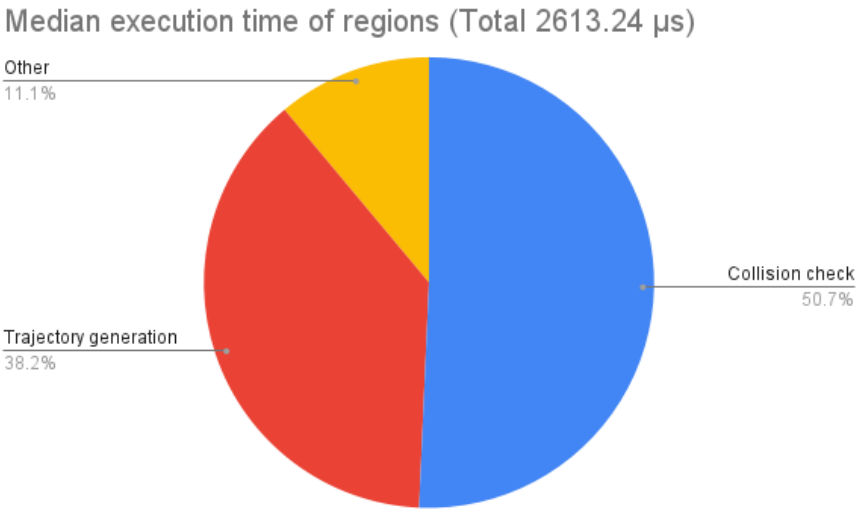

Figure 3.12: Median execution time of GPU implementation regions with benchmarking parameters listed in 3.1

4

Experimental Setup

To efficiently evaluate the effects of parallelism on the algorithm, a well-defined experimental setup must be described in terms of tools and methodologies used to benchmark it. Benchmarking requires unbiased comparisons and accurate measurements with minimal outside influence. During development, proper benchmarking is also important to identify bottlenecks and compare optimisations.

4.1 Perf profiling

Perf is a profiling tool included in the Linux kernel that can provide detailed information about performance metrics. When profiling application performance, Perf can provide valuable information about program bottlenecks. Furthermore, it can give execution time statistics on a per-function basis. Combined with tools like FlameGraph [10], the program stack trace can be visualised (see example in 4.1). The FlameGraph is a compelling visualisation that simplifies bottleneck identification. For example, in Figure 4.1, it is easily identified that the bottleneck of the program is the *generateTrajectories* function, highlighting that effort should be spent optimising this function.

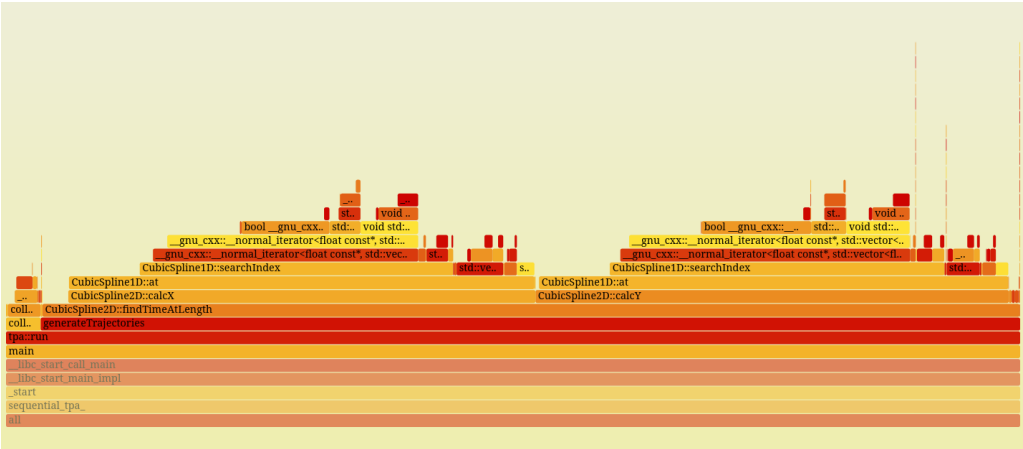

Figure 4.1: Perf stack trace data modelled using a flame graph.

4.2 C++ Chrono library

To measure the execution time of portions of programs, it can be helpful to have programming directives available that can record timestamps at specific lines of code. In C++, the chrono library provides several clocks that offer this functionality. Among these, the *high-resolution clock* provides the system's highest tick resolution. By recording timestamps before and after a code segment and then calculating the duration between the two timestamps, a high-resolution execution time measurement can be retrieved. An example of this is illustrated in 4.2, where the function call to *tpa::run* is timed using start and stop timestamps, which are then used to calculate and print the function call duration to the standard out stream.

```
auto start = std::chrono::high_resolution_clock::now();
tpa::run(current.ego, current.ref, current.csp, current.obstacles);
auto stop = std::chrono::high_resolution_clock::now();

auto duration =
    std::chrono::duration_cast<std::chrono::microseconds>(stop-start);
std::cout << duration.count() << std::endl;
```

Figure 4.2: C++ chrono library high-resolution clock usage example.

4.3 PAPI

The Performance Application Programming Interface (PAPI) [11] is a tool created by the University of Tennessee that allows developers to benchmark and evaluate application performance through an API. PAPI provides information about hardware performance counters and software-defined events. PAPI can also retrieve performance in user-defined segments, allowing for detailed performance statistics of microkernels. The metrics retrieved range from CPU performance counters to instruction count, cycle count, cache misses, and accesses. Still, it can also provide information from other components such as GPUs and interconnects [11].

To mark a section to be benchmarked through the high-level API, a region is defined as illustrated in 4.3. The events to be recorded in the defined regions are set through the *PAPI_EVENTS* environment variable. There are several events to choose from, depending on the current architecture.

```
PAPI_hl_region_begin("tpa_run");
tpa::run(current.ego, current.ref, current.csp, current.obstacles);
PAPI_hl_region_end("tpa_run");
```

Figure 4.3: Example of a PAPI high-level region definition.

When executing the compiled binary, the data collected by PAPI in these regions are exported to a JSON file where the performance counters for each execution of a

region can be analysed. PAPI also gathers information about regions executed on several threads and presents per-thread information in these situations [11].

4.4 NVIDIA Nsight

NVIDIA Nsight [12] is a toolset used for detailed system-wide performance analysis and visualise system utilisation. More specifically, NVIDIA Nsight offers pointers to locate bottlenecks and optimise and benchmark the GPU-accelerated implementation of the algorithm. NVIDIA Nsight can be leveraged to ensure an effective parallelisation and utilisation of the targeted GPU [12]. The statistics retrieved from the GPU can also be used to compare resource efficiency with CPU versions.

5

Results

This chapter comprehensively analyses the thesis results regarding execution time, trajectory quality, and performance. In short, the GPU-accelerated version is significantly faster than the sequential and multicore versions and provides lower-cost trajectories in a fraction of the execution time. This chapter also provides insights into the real-time characteristics of the algorithm implementations by analysing the standard deviations of the execution times. Finally, the GPU implementation is analysed on the kernel level to determine bottlenecks and opportunities for further optimisation.

As described in Chapter 4, the results presented in this section are gathered from a benchmarking process where each version of the algorithm (GPU-accelerated, multicore and sequential) are executed with varying trajectory, obstacle and obstacle trajectory counts. Each parameter combination is executed 50 times, each time with a different seed. Averages are then calculated for that specific parameter combination, including execution time, lowest cost and execution time standard deviation. All benchmarks are executed on a GPU-accelerated automotive computing platform.

5.1 Execution time

When analysing the execution times presented in Table 5.1 and in Figure 5.1, it is clear that the GPU-accelerated version of the algorithm is significantly faster than the sequential and multicore version, providing 2 to 3 orders of magnitude lower execution times. Furthermore, the slowest GPU-accelerated execution time shown in Table 5.2 is faster than all sequential and multicore execution times.

The execution time of the GPU-accelerated version seems to approach linear scaling between 108 and 252 trajectories, as highlighted by the slight curve seen in Figure 5.1. The scaling could be because low trajectory counts do not efficiently utilise the parallelism resources that the GPU provides, thus leaving GPU processing units underutilised. Said scaling becomes apparent when comparing the throughput and occupancy statistics presented in Tables 5.5 and 5.6 where the lower trajectory count results in an efficient use of hardware resources. The underutilisation of GPU resources for low trajectory counts would also explain the logarithmic scaling of trajectory throughput as shown in Figure 5.2. Despite this, the algorithm's GPU-accelerated version far outperforms both versions for low trajectory counts.

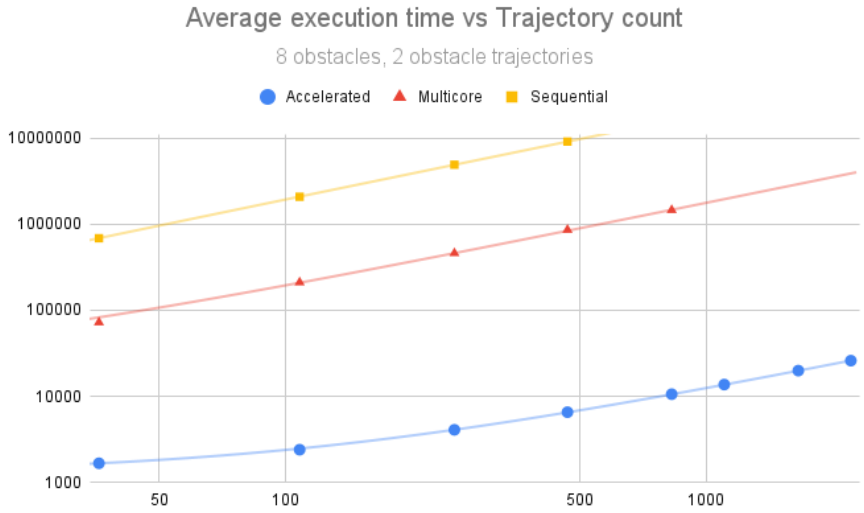

Figure 5.1: Average execution times of all algorithm versions vs the amount of trajectories generated.

Trajectories	Sequential (µs)	Multicore (µs)	GPU-accelerated (µs)	Speedup
36	686280	72267.38	1667.9	411.46
108	2078833.72	210305.86	2397.58	867.05
252	4890171.1	458418.82	4080.68	1198.37
468	9072687.16	857955.94	6538.24	1387.63
828	16024921.46	1454538	10581.22	1514.47

Table 5.1: Execution times in microseconds of the different versions of the algorithm. The speedup column shows the speedup of the GPU-accelerated version compared to the sequential version.

Trajectories	GPU-accelerated (µs)
36	1667.9
108	2397.58
252	4080.68
468	6538.24
828	10581.22
1104	13687.66
1656	19867.3
2208	25943.6

Table 5.2: Execution times in microseconds for the GPU-accelerated version of the algorithm.

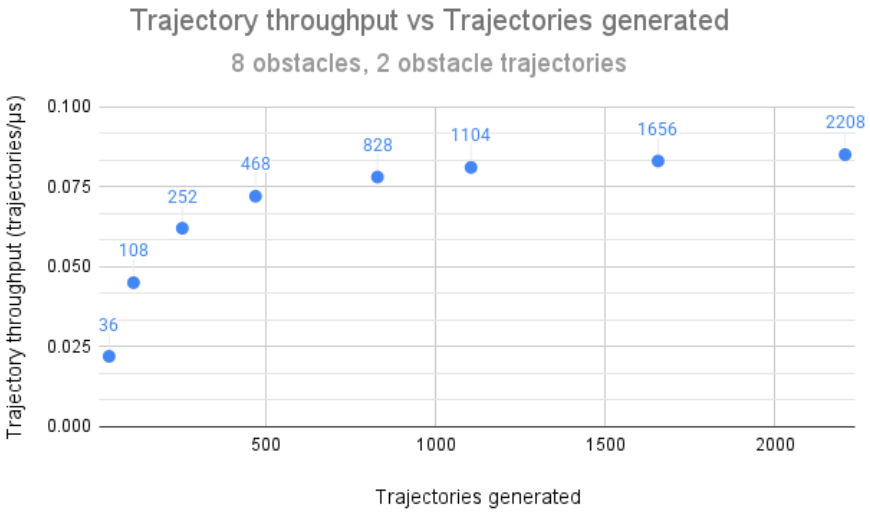

Figure 5.2: Graph of the throughput for differing amounts of trajectories generated

Introducing more obstacles to the scene seems to linearly increase the execution time of all versions of the algorithm as seen in Figures 5.3, 5.4 and 5.5. Furthermore, increasing the number of obstacle trajectories seems to result in a steeper linear scaling of the execution time.

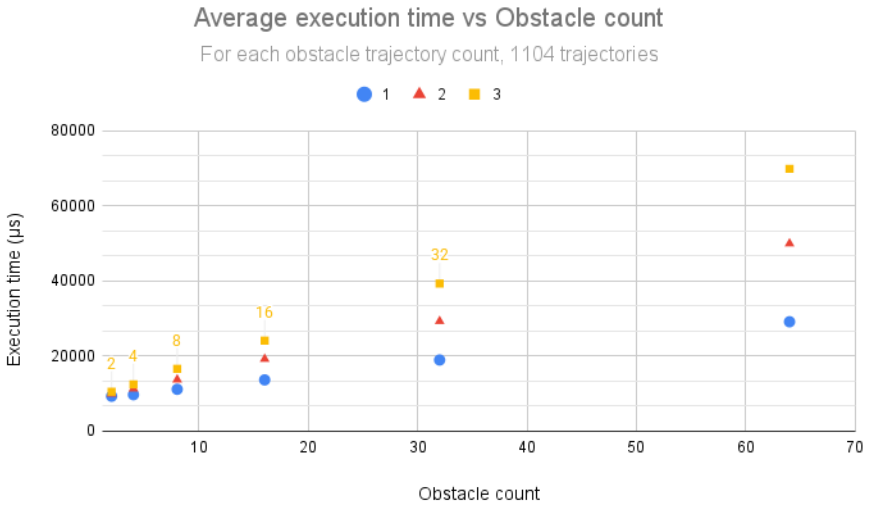

Figure 5.3: Graph of average execution time for different amounts of obstacles and obstacle paths

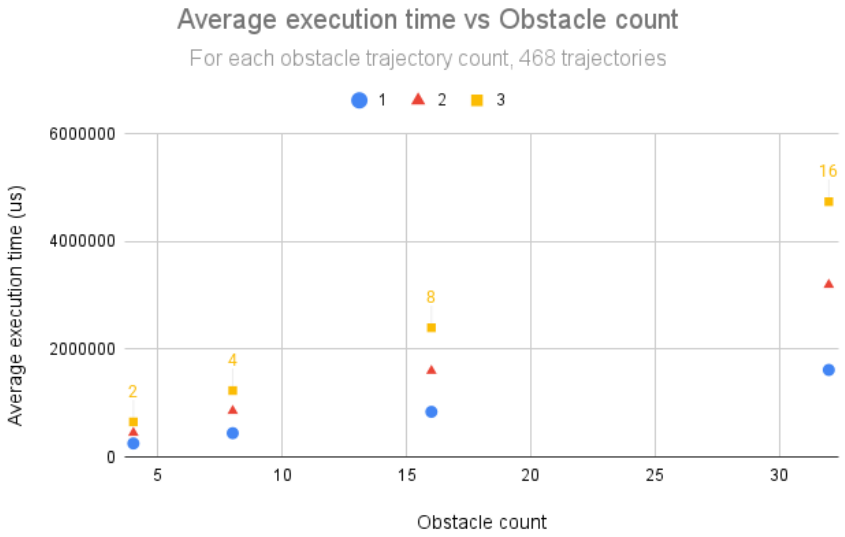

Figure 5.4: Average execution time in microseconds vs obstacle count for the multicore version of the algorithm.

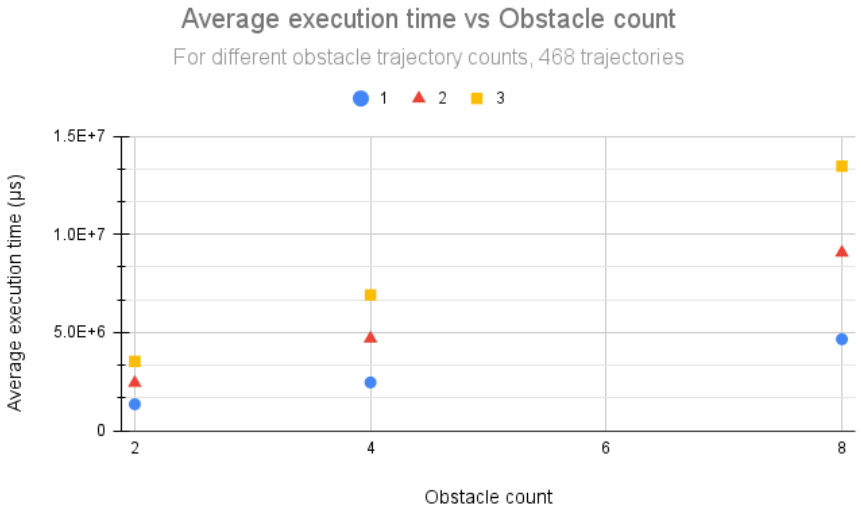

Figure 5.5: Average execution time in microseconds vs the number of obstacles for the sequential version of the algorithm.

The standard deviation of the GPU-accelerated version of the algorithm shown in Figure 5.6 proves some jitter in the execution time, especially for lower trajectory counts. The jitter may be because OS housekeeping effects disproportionately impact lower execution times than longer ones. Interestingly, the standard deviation of the execution time seems to be significantly greater for the sequential and multicore versions, as highlighted by Figures 5.8 and 5.7. Furthermore, it seems to scale linearly with the number of trajectories generated to a greater extent than the GPU-accelerated version. It could be the case that the GPU-accelerated version of

the algorithm suffers less from OS housekeeping interrupts as the computation is offloaded to the GPU, allowing the CPU to perform OS housekeeping tasks without interfering with the algorithm execution. The decreased deviation further showcases the applicability of GPUs to accelerate this algorithm in real-time scenarios.

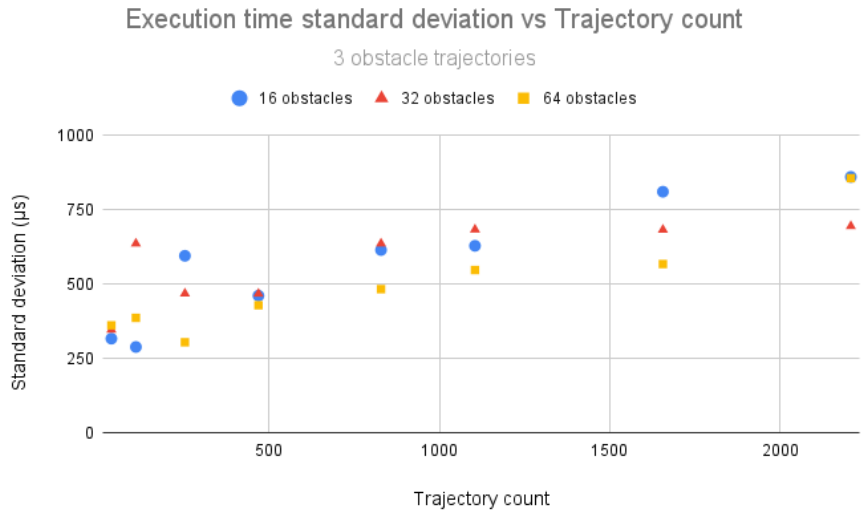

Figure 5.6: Graph of the standard deviation of average execution time vs trajectories generated.

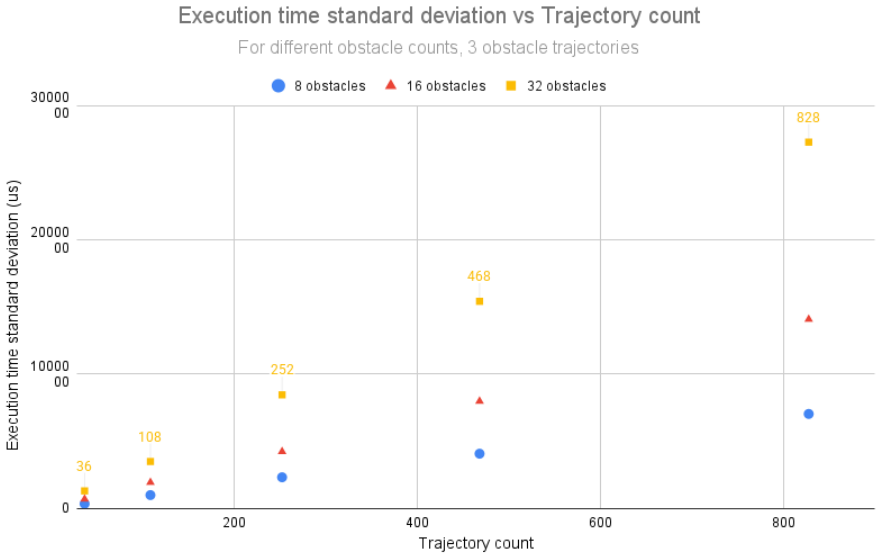

Figure 5.7: Average execution time in microseconds vs obstacle count for the multicore version of the algorithm.

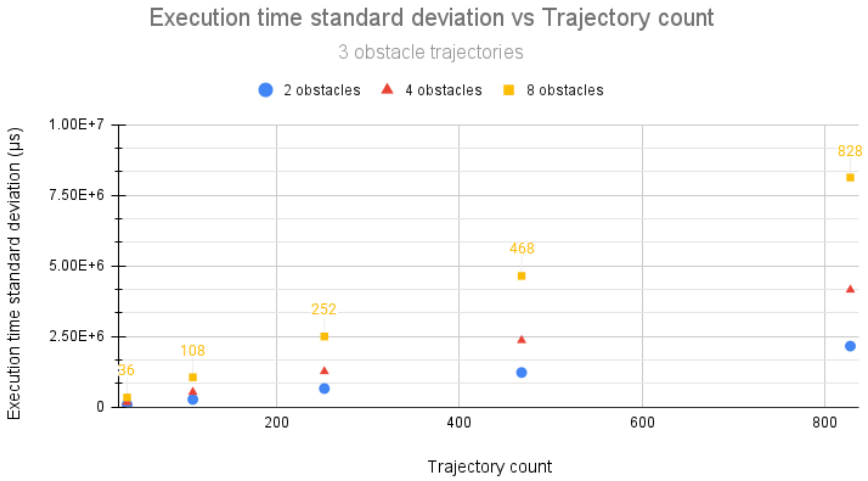

Figure 5.8: Standard deviation of the execution time in microseconds vs the number of trajectories generated for the sequential version of the algorithm. The graph shows three series for three different obstacle counts.

5.2 Trajectory quality

In the case of the GPU-accelerated algorithm, the trajectory quality gained from generating more trajectories within the solution space resulted in a notable decrease in the minimum cost as highlighted in Table 5.4. However, the cost stabilised fast, and in the case of the GPU-accelerated execution shown in Table 5.3, it did not decrease at all past 828 trajectories generated. The stabilised cost marks diminishing returns regarding trajectory quality, but it should be noted that this will depend on how the discretisation of the solution space is performed.

Trajectories	Cost normalized
36	1
108	0.2916461341
252	0.2889586996
468	0.2887225024
828	0.2876996609
1104	0.2876996609
1656	0.2876996609
2208	0.2876996609

Table 5.3: Normalised cost for the GPU-accelerated version of the algorithm for different trajectory counts. Eight Obstacles and two trajectories per obstacle were used in these benchmarks.

The multicore and sequential versions of the algorithm behave similarly when analysing the normalised costs presented in Table 5.4. However, the execution time required to obtain these costs is significantly longer.

Trajectories	GPU-accelerated cost	Multicore cost	Sequential cost
36	1	1	1
108	0.2916461341	0.3727121464	0.3727121464
252	0.2889586996	0.309484193	0.309484193
468	0.2887225024	0.2995008319	0.2995008319
828	0.2876996609	0.2945091514	0.2945091514

Table 5.4: Normalised cost of the different algorithm versions for different trajectory counts. Eight obstacles and two trajectories per obstacle were used in these benchmarks.

The same relationship as in 5.3 can be seen for all versions of the algorithm as displayed in 5.4. However, the data must be normalised to account for differences in trajectory length since the GPU-accelerated version dynamically computes the trajectory resolution to have the same number of points per trajectory. Sequential and multicore do not have this feature, and the number of points per trajectory is proportional to the target time. This feature also explains the difference in cost values between the versions. The feature was briefly ported back to the multicore and sequential versions, but resulted in worse performance which is why it was only kept in the GPU-accelerated implementation.

5.3 Kernel performance

Table 5.5 highlights that none of the kernels achieves very high memory throughput, and only the two heavy kernels responsible for sampling the trajectories and checking for collisions achieve high computational throughput. The low memory throughput implies optimisation opportunities in kernel memory accesses and how computation is performed on that memory.

The fine-grained cost computation kernel and the cost accumulation kernel are very lightweight and do not occupy the GPU for any significant period of time. Therefore, less effort was spent optimising them to avoid diminishing returns. However, alternative cost computation strategies may allow for further cuts in execution time and potentially allow for the two kernels to be merged into one.

Metric	collisionCheck	trajSampling	totalCostFG	accumCosts
Execution time (ms)	5.41	2.39	0.73	0.01
Comp. throughput (%)	65.89	84.25	6.33	0.64
Mem. throughput (%)	40.81	21.05	17.21	5.71
SM occupancy (%)	98.55	63.20	89.46	15.01

Table 5.5: Nvidia Nsight Compute statistics for each kernel in the GPU-accelerated version of the algorithm. The binary benchmark generates 1104 trajectories with eight obstacles and two obstacle trajectories.

Metric	collisionCheck	trajSampling	totalCostFG	accumCosts
Execution time (ms)	0.218	0.109	0.046	0.0087
Comp. throughput (%)	54.16	62.81	2.93	0.16
Mem. throughput (%)	33.55	16.68	9.34	0.5
SM occupancy (%)	86.87	54.76	54.53	8.74

Table 5.6: Nvidia Nsight Compute statistics for each kernel in the GPU-accelerated version of the algorithm. The binary benchmark generates 36 trajectories with eight obstacles and two obstacle trajectories.

Table 5.5 and 5.6 presents execution time, compute and memory throughput and achieved SM occupancy of each kernel. The execution time differs from the actual execution time as it is affected by being executed in a benchmarking context. The tables indicate that a lower trajectory count significantly reduces the computational throughput, memory throughput and achieved occupancy.

Extended information about Nsight Compute statistics for the trajectory sampling and collision checking kernels can be seen in Figures 5.9 and 5.10. The graphs show that resources seem unsaturated for low trajectory counts, as previously stated, and quickly stabilise and remain virtually unchanged for increasing trajectory counts. Note that for the trajectory sampling kernel, the achieved occupancy stabilises at about 66% occupancy. The occupancy is close to the theoretical maximum occupancy of this kernel that Nvidia Nsight Compute reports at 66.7% (2/3). This theoretical maximum is caused by the kernel requiring 50 registers per thread, limiting the number of warps that can run in parallel compared to the hardware maximum. Restructuring this kernel that requires fewer registers could result in significant speedups.

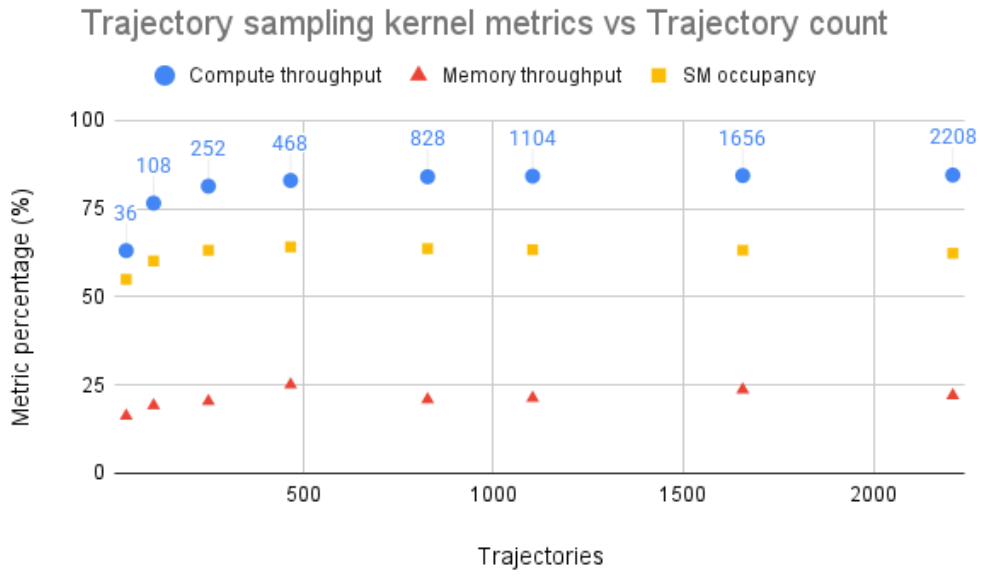

Figure 5.9: Nvidia Nsight Compute statistics for the trajectory sampling kernel vs trajectory counts. Benchmarked with eight obstacles and two obstacle trajectories.

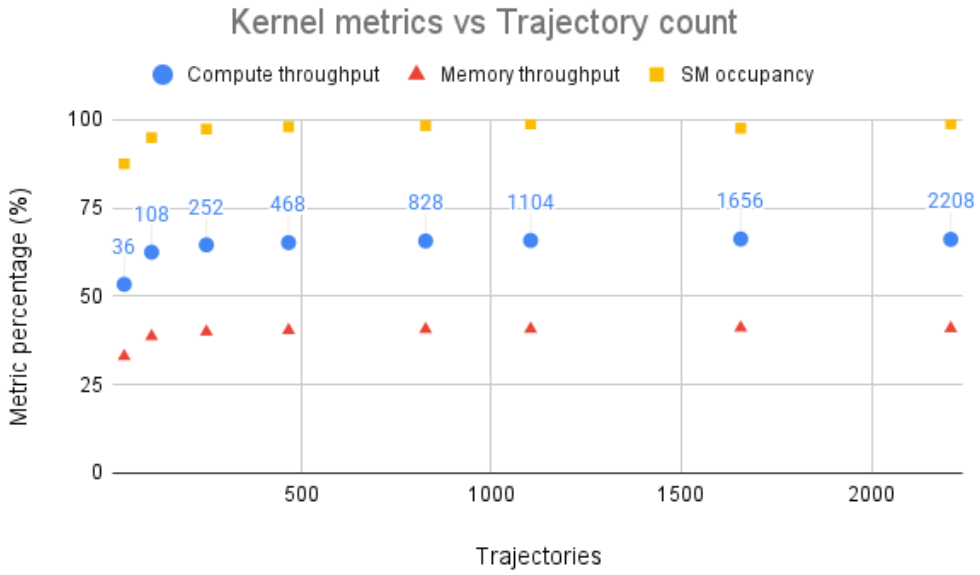

Figure 5.10: Nvidia Nsight Compute statistics for the collision checking kernel vs trajectory counts. Benchmarked with eight obstacles and two obstacle trajectories.

6

Future work

Future work within this field could include investigating further optimization strategies using CUDA techniques such as shared memory. The kernels written for this project did not allow for practical usage of this as explained in Sections 3.5.3 and 3.5.2. Nevertheless, other implementations of similar algorithms could find cases where techniques such as shared memory would be easily applicable.

An alternative cost computation strategy could also be considered as the cost computation GPU-kernels in this implementation are unoptimized and inefficient. Significant effort was not spent on optimizing these because of their already very low execution times in proportion to the other kernels. Reduction-based approaches like tree-based reduction of the costs per trajectory could be explored, which could improve performance.

Furthermore, it would also be interesting to investigate the power consumption of the different implementations of algorithms. Excessive power consumption should be avoided since they are executed on mobile platforms, and their only power supply is onboard batteries. The ability to dissipate the heat generated by the device must also be considered, as heavy computation and power consumption cause heat generation. For this reason, it might also be interesting to develop an energy-efficient version of the algorithm where resources are utilised to minimise power consumption while still meeting some performance requirements.

Another research opportunity would be to investigate the performance impact of using cross-platform libraries for GPU-acceleration, like OpenCL or SYCL, instead of CUDA. This would eliminate dependency on specific hardware and make the code portable to different platforms. It would also be meaningful to conduct a more rigid real-time analysis of GPU-accelerated algorithms within this field, mainly to investigate how GPU-acceleration affects execution time jitter in time-critical settings.

7

Conclusion

In this thesis, the trajectory generation algorithm described by Werling et al. [1] was implemented in three versions: sequential, multicore and GPU-accelerated. Initially, each version was verified for correctness using a seeded mock state generation framework, and a visualisation tool was developed alongside the algorithm implementation. The three versions were iteratively optimised while simultaneously verifying results through the tools mentioned earlier.

The algorithm versions were then benchmarked with varying trajectory count, obstacle count and obstacle trajectory count. The results highlighted that the GPU-accelerated version of the algorithm was 2 to 3 orders of magnitude faster than the sequential version. The multicore version also produced significant speedups of around one order of magnitude. Furthermore, the GPU-accelerated version proved to have a lower execution time standard deviation, indicating that is has a more stable execution time, which is more suitable for real-time scenarios. However, this is very dependent on the OS being used and should be researched further in a more rigid real-time analysis.

Furthermore, the lower execution time of the GPU-accelerated version allowed for higher resolution discretisation of the solution space, producing more trajectories to choose from, resulting in a lower minimal cost. However, the minimal cost ceased to improve after 828 trajectories, highlighting a point of diminishing returns. The diminishing returns could be attributed to how the solution space was discretised and may be significantly impacted by other discretisation strategies.

Performance analysis of the GPU-accelerated program kernels show that there is room for further optimization. The trajectory sampling and collision checking kernels seem to achieve reasonably high compute throughput, but neither seem to achieve high memory throughput. To improve this, the memory access patterns of the GPU-kernels should me analysed in more detail to find and address inefficiencies. Alternative implementation strategies of the GPU-kernels could also be investigated to achieve better performance, avoiding small computationally lightweight kernels.

Future research could investigate more advanced CUDA optimisation strategies, or different algorithm implementation approaches to increase performance further. Furthermore, the power consumption of the other algorithms could be studied since these algorithms often execute battery-powered automotive hardware, which could constrain how much power can be consumed for heat dissipation. Lastly, it would be

valuable to investigate the real-time attributes of the algorithm implementations in more detail since they often execute in time-critical scenarios where execution time jitter is of utmost importance.

Bibliography

[1] M. Werling, S. Kammel, J. Ziegler, and L. Grö ll, "Optimal trajectories for time-critical street scenarios using discretized terminal manifolds," *The International Journal of Robotics Research*, vol. 31, no. 3, pp. 346–359, 2012.

[2] K. Bergman, O. Ljungqvist, and D. Axehill, "Improved optimization of motion primitives for motion planning in state lattices," in *2019 IEEE intelligent vehicles symposium (IV)*, IEEE, 2019, pp. 2307–2314.

[3] M. McNaughton, C. Urmson, J. M. Dolan, and J.-W. Lee, "Motion planning for autonomous driving with a conformal spatiotemporal lattice," in *2011 IEEE International Conference on Robotics and Automation*, IEEE, 2011, pp. 4889–4895.

[4] N. Seegmiller, P. Barone, and E. Venator, *Motion planning in curvilinear coordinates for autonomous vehicles*, US Patent 11,884,268, Jan. 2024.

[5] T. Rauber and G. Rünger, *Parallel programming*. Springer, 2013.

[6] D. B. Kirk and W.-m. W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*, 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2010, ISBN: 0123814723.

[7] R. Strzodka, "Chapter 31 - abstraction for aos and soa layout in c++," in *GPU Computing Gems Jade Edition*, ser. Applications of GPU Computing Series, W.-m. W. Hwu, Ed., Boston: Morgan Kaufmann, 2012, pp. 429–441, ISBN: 978-0-12-385963-1. DOI: https://doi.org/10.1016/B978-0-12-385963-1.00031-9. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780123859631000319.

[8] Intel Corporation, "A guide to vectorization with intel c++ compilers," Intel Corporation, Tech. Rep., 2010. [Online]. Available: http://software.intel.com/file/31848.

[9] NVIDIA Corporation, *Cuda c++ programming guide*, Accessed: March 20, 2025, NVIDIA Corporation, 2025. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/.

[10] B. Gregg. "Flame graphs." Accessed: February 20, 2025. [Online]. Available: https://www.brendangregg.com/flamegraphs.html.

[11] ICL UTK, *Papi wiki*, https://github.com/icl-utk-edu/papi/wiki, Accessed: 2025-03-12, 2025.

[12] NVIDIA. "Nvidia nsight systems user guide." Accessed: 2025-02-25, NVIDIA Corporation. [Online]. Available: https://docs.nvidia.com/nsight-systems/UserGuide/index.html.

A

Appendix 1

This appendix includes the trajectory generation pseudocode.

```
for (lon_strategy : lon_strategies) {
    for (lat_strategy : lat_strategies) {
        frenet = cartesianToFrenet(ego_data, ref_data)
        target_times = getTargetTimes(frenet, lon_strat)

        for (t : target_times) {
            target_s = getTargetState(frenet, t, lon_strat)
            target_d = getTargetState(frenet, t, lat_strat)

            for (d = d_min; d <= d_max; d += d_step) {
                target_d_copy = target_d;
                target_d_copy.pos += d;
                dqp = QuinticPolynomial(frenet.lat, target_d_copy, t)
                for (s = s_min; s <= s_max; s += s_step) {
                    target_s_copy = target_s
                    if (lon_strategy == Strategy::VEL_KEEP) {
                        target_s_copy.vel += s
                        sqp = QuarticPolynomial(frenet.lon, target_s_copy, t)
                    } else {
                        target_s_copy.pos += s
                        sqp = QuinticPolynomial(frenet.lon, target_s_copy, t)
                    }
                }

                FrenetTrajectory fren_traj
                CartesianTrajectort cart_traj
                for (s = 0; s < t; s += sample_step) {
                    // Evaluate trajectories
                    // Convert to cartesian coordinates
                    // Save both frenet and cartesian coordinates
                }
                cost = calculateCost(fren_traj, cart_traj);
                trajectories.push_back({fren_traj, cart_traj, cost});
            }
        }
    }
}
```

Figure A.1: Trajectory generation pseudocode.