CTU

CZECH TECHNICAL UNIVERSITY IN PRAGUE

Faculty of Electrical Engineering
Department of Control Engineering

Bachelor's thesis

Hardware accelerated risk-aware motion planning

Ondřej Toman

May 2023

Supervisor: Ing. Petr Čížek
Supervisor specialist: Ing. Jakub Sláma

BACHELOR‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Toman Ondřej** Personal ID number: **498998**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Control Engineering**

Study program: **Cybernetics and Robotics**

II. Bachelor’s thesis details

Bachelor’s thesis title in English:

Hardware accelerated risk-aware motion planning

Bachelor’s thesis title in Czech:

Hardwarově akcelerované rizikové plánování ve scénářích městské vzdušné mobility

Guidelines:

1) Get familiar with development for FPGA System on a Programmable Chip boards such as DE10 nano [1] with a focus on the High Level Synthesis using C code [2].
2) Get familiar with the risk-aware trajectory planning for unmanned air vehicles and its implementation [3].
3) Benchmark the baseline CPU-based implementation of the risk-aware trajectory planning, identify parts suitable for hardware-acceleration and propose implementation on the FPGA.
4) Benchmark the performance of the FPGA implementation in comparison to the CPU-based implementation.

Bibliography / sources:

[1] DE10 nano get started guide, available:
https://software.intel.com/content/www/us/en/develop/articles/terasic-de10-nano-get-started-guide.html [cited on 2023-26-01]
[2] Intel High Level Synthesis Compiler Pro Edition - User Guide, available:
https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/hls/ug-hls.pdf
[cited on 2023-26-01]
[3] J. Sláma, P. Váňa, and J. Faigl: Risk-aware Trajectory Planning in Urban Environments with Safe Emergency Landing Guarantee, IEEE International Conference on Automation Science and Engineering (CASE), 2021, pp. 1606-1612.

Name and workplace of bachelor’s thesis supervisor:

Ing. Petr Čížek Department of Computer Science FEE

Name and workplace of second bachelor’s thesis supervisor or consultant:

Date of bachelor’s thesis assignment: **06.02.2023** Deadline for bachelor thesis submission: **26.05.2023**

Assignment valid until: **22.09.2024**

Ing. Petr Čížek	prof. Ing. Michael Šebek, DrSc.	prof. Mgr. Petr Páta, Ph.D.
Supervisor’s signature	Head of department’s signature	Dean’s signature

III. Assignment receipt

The student acknowledges that the bachelor’s thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the bachelor’s thesis, the author must state the names of consultants and include a list of references.

.	
Date of assignment receipt	Student’s signature

© ČVUT v Praze, Design: ČVUT v Praze, VIC

Declaration

I declare that the presented work was developed independently and that I have listed all sources of the information used within it in accordance with the methodical instructions for observing the ethiclal principles in the preparation of university theses.

Prague, May 26, 2023

. . . . . .

Ondřej Toman

Acknowledgement

First of all, I would like to thank my supervisor Ing. Petr Čížek for his feedback and guidance during the writing of this thesis. I am also grateful to my family for supporting me during my studies.

Abstract

The Risk-based RRT* algorithm is used for the trajectory planning for aerial vehicles to minimize the damage caused by a potential crash. However, a major limitation of this algorithm is its computational complexity. In this thesis, we focus on hardware acceleration of the algorithm using the Field-programmable Gate Array (FPGA). First, we theoretically analyze the computational complexity of the different parts of the algorithm. We verify the conclusions of the analysis by benchmarking the existing Julia and developed C++ implementations of the algorithm and identify the crash risk calculation as the main performance bottleneck of the existing, purely CPU-based, implementation of the algorithm. Next, we design and benchmark a custom developed FPGA component for hardware acceleration of the crash risk calculation that is integrated within the System on a Programmable Chip (SoPC) design of the developed architecture. Based on the properties of the developed component and its benchmarking results, we propose an SoPC FPGA architecture for acceleration of the Risk-based RRT* algorithm.

Keywords: Risk-based RRT*, risk planning, hardware acceleration, FPGA

Abstrakt

Algoritmus Risk-based RRT* umožňuje plánovat trajektorie bezpilotních letounů tak, aby byly minimalizovány škody způsobené případnou havárií. Možnost reálného nasazení a tím i zvýšení bezpečnosti leteckého provozu však omezuje relativně značná výpočetní náročnost tohoto algoritmu. Proto jsme se v této práci zaměřili na možnosti hardwarové akcelerace pomocí technologie Field-programmable Gate Array (FPGA). Nejprve jsme teoreticky analyzovali výpočetní náročnost jednotlivých částí původního algoritmu. Závěry této analýzy jsme ověřili měřením, nejprve na již existující implementaci v jazyce Julia, později i na vyvinuté implementaci v jazyce C++ přímo na vývojové FPGA desce DE10-Nano. Na základě naměřených hodnot jsme jako nejnáročnější operaci vyhodnotili výpočet míry rizika daného plánovacího segmentu. Pro urychlení tohoto výpočtu jsme navrhli a otestovali FPGA komponentu maticového násobení po prvcích. Nakonec jsme na základě vlastností komponenty navrhli systémovou architekturu obvodu pro zrychlení kompletního plánovacího algoritmu Risk-based RRT*.

Klíčová slova: Risk-based RRT*, rizikové plánování, hardwarová akcelerace, FPGA

Contents

Acronyms

CPU Central Processing Unit

DMA Direct Memory Access

EWMM Element-wise Matrix Multiplication

FPGA Field-programmable Gate Array

HDL Hardware Description Language

HLS High-level Synthesis

HPS Hard Processor System

NNS Nearest Neighbor Search

RRT Rapidly-exploring Random Tree

RRT* Asymptotically optimal RRT

SoPC System on a Programmable Chip

UAV Unmanned Aerial Vehicle

List of Figures

List of Tables

List of Algorithms

Chapter 1
Introduction

With the advent of Unmanned Aerial Vehicles (UAVs) and their anticipated use in transporting packages or people in cities, a new risk is emerging. With large number of UAVs moving over cities, the likelihood of a crash increases. This increases not only the risk of destroying the aircraft itself, but also the risk of injury to persons present at the point of impact.

This risk cannot be eliminated completely, but it can be minimised. We can choose a flight path that minimises the risk of damage in the case of a crash. The risk-based Asymptotically optimal RRT (RRT*) algorithm [1] is based on this principle. However, as mentioned in [2], the time to calculate a trajectory using this algorithm can reach hundreds of seconds in some cases. Moreover, these values were measured on a desktop processor. Thus, we can expect that when deployed on small UAVs with limited computational resources the result will be even worse. Waiting several minutes to calculate the trajectory before actual take-off can pose a logistical problem when transporting packages. When transporting people, this would mean an increase in flight time. The current algorithm also takes into account only ballistic fall in assessment of the trajectory risk, while there are many different causes of UAV failure [1]. Therefore, it is desirable to optimise the speed of the algorithm.

Various software accelerations of the underlying sampling based planning have been presented in the literature [3–5]. Thus, finding further ways of software optimization could be difficult. Instead, we focus on hardware acceleration of the algorithm using the Field-programmable Gate Array (FPGA) architecture. We approach the problem by as follows. First, we need to determine which parts are the slowest and in which cases hardware acceleration is beneficial. Nowadays FPGA systems allows for efficient division of the computation between the parts that are beneficial to accelerate using custom-designed hardware, and parts that are too costly to implement in hardware through the SoPC design. The SoPC design combines the benefits of both the general purpose Central Processing Unit (CPU)-based architecture and highly optimized FPGA circuits. The architectures communicate through bridges, that allow them to seamlesly transfer data and results between each other. However, a clever approach is necessary to the design of the SoPC systems, as communication with the hardware component requires additional arbitrage, which could slow down the computation more than the original software implementation.

Moreover, as the RRT* algorithm is asymptotically optimal, the longer the computation takes, the better trajectory is found. Thus, if we could speed up significantly a certain part of the computation, we would not only get a faster result with less energy consumption, but also the possibility of finding a better solution to the problem in a given time.

The thesis is structured as follows. The principle of RRT* and the risk-based RRT* developed from it are described in Chapter 2, together with the identification of the computationally intensive parts and analysis of the possibilities of hardware acceleration using FPGA/Hard Processor System (HPS). Theoretical analysis of the algorithm is performed in Chapter 3, followed by the practical benchmarking at the beginning of Chapter 4. Next, design and benchmark of a component to compute Element-wise Matrix Multiplication (EWMM), which we identify as the most computationally expensive operation in trajectory risk computation, is presented in Chapter 4. Finally, based on the identified properties of this component, we propose an architecture that should provide an acceleration of the whole algorithm due to a series of optimizations, in Chapter 5.

Chapter 2
Background

Our work is focused on the hardware acceleration of the risk-based RRT* algorithm. Therefore, in the first part of this chapter we describe the principles of the Rapidly-exploring Random Tree (RRT) algorithm and its variants. Next, we explain the risk-based RRT* algorithm which we build upon in the following chapters. In the last part, we introduce the FPGA technology and the High-level Synthesis (HLS) method, which we will use to test our designed circuit.

2.1 RRT and its Variants

RRT [6] is a path-finding algorithm based on random sampling of the configuration space. It is based on the principle of covering a continuous space with a graph to find a path from the starting configuration to the goal configuration. The goal of the algorithm is to cover the search space with a tree where a path to the specified goal can be found using backtracking.

The algorithm starts with a given initial configuration q_{init}, a maximum edge length Δ_{step} and the number of iterations N. The tree is generated by repeating four steps - sample, Nearest Neighbor Search (NNS), steer and configuration adding. First, a graph G containing the configuration q_{init} is created. Next, in the sample step, a configuration q_{rand} is randomly selected from the search space. If we also know the goal configuration of the path at the time of running the algorithm, we can affect this selection by choosing the probability distribution from which the point is generated. In the basic version of RRT, a uniform distribution is used, but in case we have expert knowledge of the planning problem, choosing a different distribution may lead to faster path finding.

The next step is the NNS, in which the closest configuration q_{near} from the graph G to the point q_{rand} is found. The NNS itself is a non-trivial operation. In practice, this step can be implemented using the k-d tree algorithm [7].

Once the closest node in graph is found, the steering step is performed. A new configuration q_{new} is created such that it lies on the curve between the vertices q_{near} and q_{rand}, while at the same time it is located at most at distance Δ_{step} from q_{near} and the space between q_{new} and q_{near} along this curve is free. The curve can be a line segment, or a more general curve can be choosed, depending on the application. This case is further discussed in the Section 2.1.3 as the risk-based RRT* uses aircraft motion model to model the curve [2].

Finally, the configuration q_{new} and the edge $(q_{\text{new}}, q_{\text{near}})$ are added to the graph G. After N iterations, we obtain a tree covering the search space. The larger N we choose, the better coverage of the space we get. If the path goal configuration is known during the algorithm run, the algorithm terminates as soon as a configuration close enough to the goal configuration is added. To find a path in this tree, we only need to find the closest configuration of the graph to the given goal configuration and follow the parents until the initial configuration is reached. The pseudocode of the algorithm is listed in Algorithm 1 and one step of the algorithm is visualized in Figure 2.1.

The advantage of RRT over standard planning algorithms is its efficiency in high-dimensional configuration spaces. Another advantage is that once the configuration space is covered by the graph, finding a path from this initial configuration to the various targets is almost instantaneous. However, once the initial configuration q_{init} is changed, a new graph must be generated. The narrow corridors in the configuration space are difficult for the RRT to pass by random sampling. This problem is partially

Algorithm 1 RRT

Require: q_{init} – Initial configuration
Require: Δ_{step} – Maximum edge length
Require: N – Number of iterations
Ensure: G – RRT graph

```
G.init(q_init)
for i = 1 to N do
    q_rand ← Sample()
    q_near ← Nearest(q_rand, G)
    q_new ← Steer(q_near, q_rand, Δ_step)
    G.AddConfiguration(q_new)
    G.AddEdge(q_near, q_new)
end for
return G
```

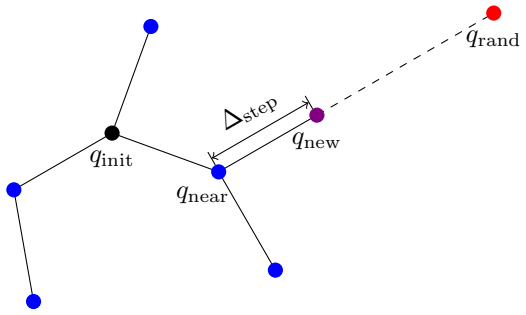


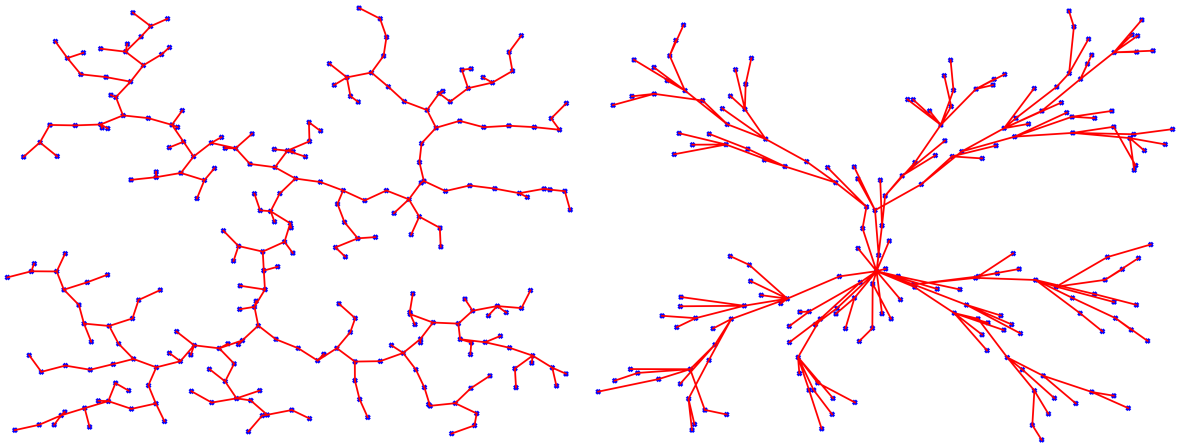
Figure 2.1: One step of RRT. First, the point q_{rand} is generated. The closest configuration q_{near} to it is found. At a distance Δ_{step} from q_{near}, a new configuration q_{new} is created and added to the tree.

solvable by choosing a different probability distribution from which the q_{rand} point is generated. By doing so, the probability of generating a point in this narrow spot can be increased, allowing the space to be covered faster [8]. However, this modification can also have the opposite effect. RRT also does not include any mechanism for optimizing the path found, which is the main disadvantage of this algorithm, but path finding itself may be sufficient in hard-to-solve problems.

2.1.1 RRT*

To find an asymptotically optimal path, an RRT* [9] can be used, which also optimizes the connections between vertices as the tree is expanded. The comparison of RRT and RRT* is shown in Figure 2.2 When a configuration q_{new} is added, all vertices in the Δ_{step} neighborhood are found.

For these vertices, it is tested whether linking them through the newly generated configuration creates a shorter path to the initial configuration. If so, the node is reconnected. The L2-norm is usually used to calculate the distance, but other variants of RRT* may calculate the distance in a different way. In the case of an RRT with a specified goal configuration, the algorithm is usually terminated when the path is found, since a longer run of the algorithm will not improve the path found. In the case of RRT* with a specified goal configuration, where the path is optimized when a configuration is added, a longer run of the algorithm makes sense. This modification makes the RRT* asymptotically optimal. Thus, the longer the algorithm runs, the closer the found path will be to the optimal solution. Although the RRT* is asymptotically optimal, it can be computationally expensive. Thus, further variants of RRT-based algorithms have been proposed to improve their performance.



(a) RRT tree (b) RRT* tree

Figure 2.2: Comparison of RRT tree (a) and RRT* tree (b). While RRT only builds the search tree, RRT* adjusts the connections between the tree vertices to optimize the path found. This makes RRT* asymptotically optimal. Generated using [10].

2.1.2 Improvements of RRT*

RRT-based algorithms are widely used in robotics for their efficiency in high-dimensional spaces. However, since robots and drones are typically small battery-powered devices with limited computational power, it is important to look for more efficient algorithms. One of those improvements to RRT* is the Informed RRT* [3], which uses heuristics to converge faster to an optimal solution. This variant starts the same way as RRT* until a path to the goal is found. Then, the selection of random points from the space is restricted to a region defined by an ellipse, where the start and goal points serve as focal points and the length of the found path corresponds to twice the length of semi-major axis. This ensures that points that cannot form a shorter path than the one already found will not be selected. Once a better path is found the ellipse is recalculated.

Another option is for example Real-Time RRT* [11]. This is an adaptive variant of RRT* that allows the search tree to change dynamically as the starting position or obstacle movement changes. This variant can be useful when exploring unfamiliar terrain where frequent planning is required as new terrain is revealed.

2.1.3 Risk-based RRT*

A shortest path is the typical goal of a path planning task, and so a distance function is used as a cost function in RRT-based algorithms. However, in certain specific path planning problems, it may be desirable to optimize different cost function. One example is a risk-based RRT* variant [2]. This variant is proposed for small aircraft flying over cities and minimizes the number of casualties in the case of in-flight failure. Furthermore, the risk increases with the length of the flight, so it also naturally provides short trajectories but avoids areas with high population densities. Pseudocode of this algorithm is listed in algorithm 2.

The input parameters are initial and target configurations, terrain altitude map, and no-fly zone map.

First, based on the terrain altitudes, aircraft dynamics and locations of the landing sites, a safe altitude map is generated to guarantee a safe landing under loss of thrust [12]. This is followed by tree expansion, which unlike RRT* starts from the target configuration q_{f}. The tree expansion process starts by randomly selecting the configuration q_{rand} from this space. The nearest neighbor to this configuration q_{near} in the sense of the length of Dubins maneuver is found [13]. In the steer step, a

Algorithm 2 Risk-based RRT* (inspired by [2])

Require: q_{init} – Initial configuration of the aircraft
Require: q_{f} – Final configuration of the aircraft
Require: $\mathcal{T}_{\text{alt}}$ – Altitude of the terrain (or obstacles)
Require: $\mathcal{Z}$ – Map of no-flight zones
Ensure: Γ – The least risky trajectory
Ensure: $\mathcal{R}(q_i)$ – Risk of trajectory Γ

1: $G_l, \mathcal{A} \leftarrow \text{SafeLandingMap}(\Xi, \mathcal{T}_{\text{alt}})$
2: $G \leftarrow (\boldsymbol{V} \leftarrow q_f, \boldsymbol{E} \leftarrow \emptyset)$
3: $\mathcal{R}(q_f) \leftarrow 0$
4: **repeat**
5: $\quad q_{\text{rand}} \leftarrow \text{Sample}()$
6: $\quad q_{\text{near}} \leftarrow \text{Nearest}(q_{\text{rand}}, G)$
7: $\quad q_{\text{new}} \leftarrow \text{Steer}(q_{\text{near}}, q_{\text{rand}})$
8: $\quad Q_{\text{near}} \leftarrow \text{Near}(q_{\text{new}}, G)$
9: $\quad q* \leftarrow \text{argmin}_{q_{\text{n}} \in Q_{\text{near}}} [\mathcal{R}(q_{\text{n}}) + \mathcal{R}(q_{\text{new}}, q_{\text{n}})]$
10: $\quad \mathcal{R}(q_{\text{new}}) \leftarrow \mathcal{R}(q*) + \mathcal{R}(q_{\text{new}}, q*)$
11: $\quad$ **if** $\text{isAdmissible}((q_{\text{new}}, q*), G_l, \mathcal{A}, \mathcal{T}_{\text{alt}}, \mathcal{Z})$ **then**
12: $\quad\quad \boldsymbol{V} \leftarrow \boldsymbol{V} \cup \{q_{\text{new}}\}$
13: $\quad\quad \boldsymbol{E} \leftarrow \boldsymbol{E} \cup \{(q*, q_{\text{new}})\}$
14: $\quad\quad G \leftarrow \text{Rewire}(Q_{\text{near}}, G)$
15: $\quad$ **end if**
16: **until** $||q_{new} - q_i|| \leq \Delta_{\text{tol}}$
17: $Q_{\text{near}} \leftarrow \text{Near}(q_i, G)$
18: $q* \leftarrow \text{argmin}_{q_{\text{n}} \in Q_{\text{near}}} [\mathcal{R}(q_{\text{n}}) + \mathcal{R}(q_i, q_{\text{n}})]$
19: $\mathcal{R}(q_i) \leftarrow \mathcal{R}(q*) + \mathcal{R}(q_i, q*)$
20: **if not** $\text{isAdmissible}((q_i, q*), G_l, \mathcal{A}, \mathcal{T}_{\text{alt}}, \mathcal{Z})$ **then**
21: $\quad$ **goto** Line 4
22: **end if**

new configuration q_{new} is created, at maximum distance Δ_{step}. Since the configuration q_{near} has been selected using the length of Dubins maneuver, the edge $(q_{\text{new}}, q_{\text{near}})$ may not be an ideal edge in the meaning of risk. Since computing the risk is computationally more demanding than computing the length of Dubins maneuver, the set of k nearest neighbors Q_{near} in the sense of the length of Dubins maneuver is first found. We assume that the risk of an edge will be correlated with its length. Now the best parent $q*$ of a configuration q_{new} is selected from this small set, this time according to the risk. If the edge $(q*, q_{\text{new}})$ is admissible, the configuration q_{new} with this edge is added to the graph. To ensure optimality, edges in the vicinity of configuration q_{new} are rewire in the rewire step to minimize the risk to the goal. Tree expansion continues until the distance from the initial configuration is less than Δ_{tol}. Finally, the initial configuration itself is added to the graph and by following the parents towards the target configuration, the resulting trajectory is retrieved.

Since this algorithm is used to plan the trajectory of the aircraft, it is necessary to respect its dynamic constraints when planning. These constraints can be modelled by a Dubins airplane model

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{z} \\ \dot{\theta} \end{bmatrix} = v \begin{bmatrix} \cos\theta\cos\psi \\ \sin\theta\cos\psi \\ \sin\psi \\ u_\theta \rho^{-1} \end{bmatrix}, \tag{2.1}$$

where v is the speed of the aircraft, u_θ is the control input affecting heading angle, and ρ denotes the

minimum turning radius [2].

The edge admissibility check is performed by sampling the trajectory with sample distance d_{step}. An edge is marked as admissible if none of the points intersects the no-fly zone and also is not below the minimum safe height. Edges evaluated as inadmissible are rated with infinite risk.

The risk of an edge is defined as the integral of the risk in each aircraft configuration along trajectory Γ multiplied by the probability of aircraft failure. This can be described by equation

$$\mathcal{R} = p_{\text{fail}} \int_0^T \mathcal{M}(\Gamma(t))\mathrm{d}t, \tag{2.2}$$

where the configuration risk is then defined as

$$\mathcal{M}(q) = \int_{\mathbb{R}^2} p_{\text{imp}}(\mathbf{x}|\Gamma_{\text{bal}}) M(\mathbf{x}, E, \gamma)\mathrm{d}\mathbf{x}, \tag{2.3}$$

where $p_{\text{imp}}(\mathbf{x}|\Gamma_{\text{bal}})$ is the probability of impact and $M(\mathbf{x}, E, \gamma)$ denotes ground risk. γ is the angle of impact. The location x and energy E of impact can be determined from the description of the ballistic curve described by equation

$$m\dot{\mathbf{v}} = m\mathbf{g} - \frac{1}{2}c\rho S||\mathbf{v}||\mathbf{v}. \tag{2.4}$$

Ground risk is defined as

$$M(\mathbf{x}, E, \gamma) = p_{\text{hit}}(\mathbf{x}, \gamma)\, p_{\text{casualty}}(\mathbf{x}, E), \tag{2.5}$$

where $p_{\text{hit}}(\mathbf{x}, \gamma)$ is the probability of hitting a person and $p_{\text{casualty}}(\mathbf{x}, E)$ is the probability of casualty. $p_{\text{casualty}}(\mathbf{x}, E)$ is described by equation

$$p_{\text{hit}}(\mathbf{x}, \gamma) = \rho(\mathbf{x}) A_{\text{exp}}(\gamma), \tag{2.6}$$

where $\rho(\mathbf{x})$ denotes the population density at the point of impact and the size of the exposed area $A_{\text{exp}}(\gamma)$. Exposed area is described by equation

$$A_{\text{exp}}(\gamma) = 2(r_{\text{p}} + r_{\text{uav}})\frac{h_{\text{p}}}{\tan(\gamma)} + \pi(r_{\text{p}} + r_{\text{uav}})^2, \tag{2.7}$$

where h_{p}, r_{p} and r_{uav} denotes the height and radius of the average person and the radius of the aircraft. $p_{\text{casualty}}(\mathbf{x}, E)$ is defined as

$$p_{\text{casualty}}(\mathbf{x}, E) = \frac{1-k}{1-2k+\sqrt{\frac{\alpha}{\beta}}\left(\frac{\beta}{E}\right)^{\frac{3}{S(\mathbf{x})}}}, \tag{2.8}$$

where $S(\mathbf{x})$ is the shelter factor, α is the impact energy required to achieve $p_{\text{casualty}} = 50\,\%$ for $S = 6$ and β is the impact energy causing the accident for $S \to 0$ [2]. k is then defined as

$$k = \min\left(1, \left(\frac{\beta}{E}\right)^{\frac{3}{S(\mathbf{x})}}\right). \tag{2.9}$$

2.2 Field-programmable Gate Array

FPGA is a programmable digital circuit that allows to implement arbitrary combinational and sequential logic circuits based on software description. This is achieved by a large number of programmable blocks that are capable of implementing basic logic operations. Each block contains a look-up table, a multiplexer and a flip-flop circuit as shown in Figure 2.3 which shows a logic block structure of the

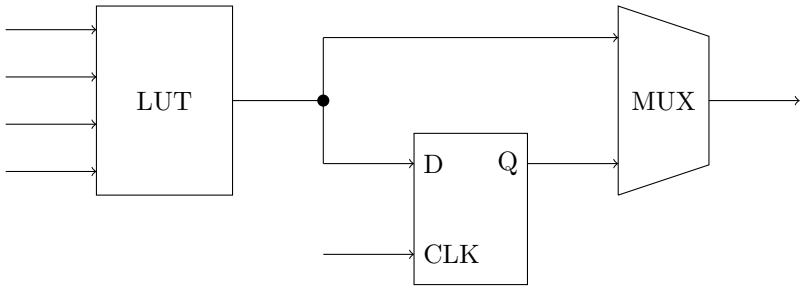


Figure 2.3: One FPGA block of the Intel Cyclone V FPGA. The multiplexer interconnects the look-up table and the D flip-flop circuit. By interconnecting multiple blocks using interconnecting segments, any sequential circuit can be implemented.

Intel Cyclone V architecture. Any logic function with up to four inputs can be implemented using the look-up table. By combining multiple blocks with flip-flop circuits, memory circuits can be created and the multiplexer can be used to switch between these operations. These blocks can be connected using interconnecting segments to create various logic circuits.

In addition to the programmable blocks, the FPGA contains several frequently used circuits that are embedded as classical unchangeable circuits because of the efficiency and size of the corresponding circuits built using programmable blocks, such as communication endpoints, memory blocks, multipliers, and phase lock loops [14].

The process of building an FPGA circuit consist of multiple phases. First, the functionality of the circuit has to be defined using the Hardware Description Language (HDL) code. Through the proces of synthesis, the HDL code is converted into the desired gait circuit. The gait circuit is converted into programmable blocks during mapping and the programmable blocks are placed in specific regions in the FPGA during the place and route phase. Finally, the result is converted into a bitstream, which can be loaded into the FPGA. This bitstream is used to configure the individual blocks and their interconnections.

FPGAs represent an option in hardware development that brings a number of advantages. One of the main advantages of the FPGAs is that there is no need to manufacture expensive custom chips to create a custom circuit. This is advantageous for small series production where making custom chips does not make economic sense. Compared to software emulation, an FPGA is much faster and more economical. However, developing a hardware implementation is more challenging and more time consuming than a software implementation for the CPU. Therefore, FPGAs are combined with HPS containing traditional CPU. This allows to divide certain problems between the software part and hardware part and exploit the benefits of both architectures. These combined systems are called SoPC. Thus, while FPGA is suitable for real-time data processing due to its low latency and parallel processing, HPS can perform sequential operations that are not worth implementing on FPGA.

2.2.1 High-level Synthesis

In this thesis, we design a component on FPGA to speed up the risk-based RRT* algorithm. To make the design of the circuit easier, we use the HLS method. Unlike HDL languages, HLS allows us to describe the circuit using a high-level programming language, usually C or C++. This allows the circuit to be described at a higher level of abstraction, simplifying and speeding up the circuit design process. The main advantage is that when the design is modified, the architecture is automatically adjusted to optimize the circuit for the application. However, this also means that the designer has less control over the architecture of the synthesized circuit.

Chapter 3
Performace analysis

In this chapter, we theoretically evaluate the computational complexity of each part of the RRT and risk-based RRT* algorithms. In the first part, we examine the RRT algorithm in detail and determine which of the sample, nearest neighbor search, steer, and configuration adding steps represent the major bottleneck of the algorithm. We also describe some methods for accelerating these steps. In the second part, we focus on the risk-based RRT* algorithm, emphasizing its differences from the basic RRT variant.

3.1 Bottlenecks of RRT Algorithm

RRT algorithm is composed of four general steps - sample, nearest neighbor search, steer, and configuration adding. Changing this structure would imply the creation of a new algorithm, which is not the goal of this thesis. Thus, to speed up the algorithm, we will look into the possibility of speeding up these subproblems.

We first examine the sample and configuration adding steps. In the sample step, a new point in the state-space is sampled. This operation can be accelerated by using more efficient pseudorandom number generation algorithms, but this may result in non-uniform sampling. Since this operation generally has constant time complexity, the speedup obtained by this modification is negligible. When adding a new configuration to a graph, it depends mainly on the representation of the graph structure in the memory. Using a different data structure may be more efficient, even at the cost of slower configuration addition to the graph. Typically, the graph is represented by structures with constant time complexity of adding a configuration, and thus it does not make sense to improve this step.

Finding nearest neighbour is a well-known non-trivial problem. Therefore, a number of solutions have emerged. A straightforward solution is a linear search, i.e., compute the distance to each point in the search set and select the point with the smallest distance. This solution has a time complexity of $\mathcal{O}(n)$, but in the case of RRT-based algorithms in which this computation occurs repeatedly on an ever-increasing set of points, this approach is impractical.

A better solution may be to use the k-d tree structure. As new points are added, they are arranged in a binary tree according to their position in space. Each point divides the subspace in which it is located into two parts according to the coordinate determined by its depth in the tree. Thus, if we assume a two-dimensional space and the root node divides the space according to the x-axis, then its descendants divide these parts of the space according to the y-axis. The descendants of those descendants again divide that part of space along the x-axis, and so on. In such an ordered set of points, finding the nearest neighbor is simple. We just use binary search to find the location of a point in the tree and at each step calculate the distance between the point and present configuration. If the distance is shorter than the distance found so far, we declare this point to be the nearest. This gives us an approximate solution to the problem. We can now discard all branches that cannot contain a closer point, allowing us to easily reject large groups of points. Finally, we verify the distance for the remaining points, which will accurately determine the nearest neighbor. This solution has the advantage of using a binary tree, which makes the time complexity only $\mathcal{O}(\log n)$ [7]. Figure 3.1 shows a visualization of the k-d tree for two dimensional data.

Similarly, we can use an R-tree structure in which points are partitioned into a hierarchy of nested

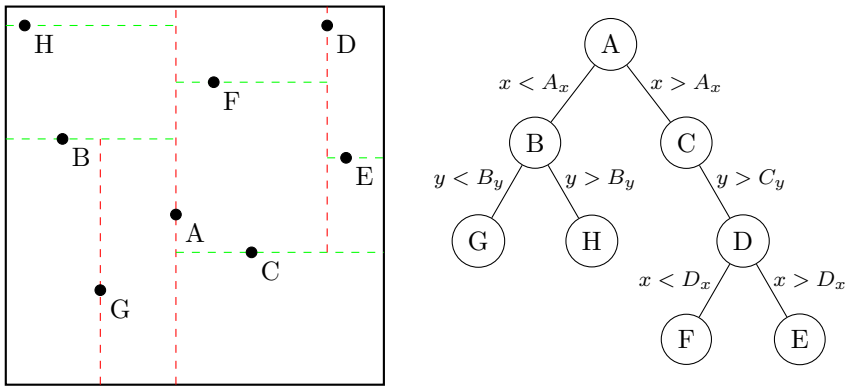


Figure 3.1: Example of k-d tree structure for two-dimensional configuration space. Each point of the tree divides the space into two parts. The first floor of the tree divides the space along the x-axis, the second along the y-axis, and so on.

rectangles. Nearby points are enclosed in rectangles, nearby rectangles are enclosed in larger rectangles, and so on. Finally, we obtain a tree structure of nested rectangles. This ensures that the nearest neighbor is likely to be in the same rectangle as the reference point. This allows us to easily exclude large groups of points that cannot be nearest neighbors [15].

Since RRT*-based algorithms include additional NNS calculations in the rewire step, the efficiency of this algorithm has an even greater impact on performance. Another issue can be frequent computation of the norm. In variants that use a more complicated relation instead of the standard L2-norm, even with efficient NNS algorithms, the complexity of this step can be high.

The last, also non-trivial problem, is the steer step, in which a new configuration in the graph is moved along the curve, between the new point and the closest configuration in the graph. This step is mainly used to detect collisions with obstacles. The simplest solution is to sample the connecting curve at several points where collisions will be checked. However, regardless of the sampling density, this solution does not guarantee a collision-free edge. For robust collision detection, the obstacles in the space must be made artificially larger, by more than half the sampling distance, and the curve must be linear between these points. This ensures that if a pair of adjacent samples does not collide with an obstacle, the edge between these samples cannot collide. While this solution ensures detection, it may cause false collisions due to artificially enlarged obstacles [8].

Another option is to represent obstacles using polygons, which allows for linear edges to detect collisions analytically. Since the polyhedrons can be complicated or high-dimensional, methods that speed up detection are used in practice. One of these methods is using bounding boxes [16]. By bordering the polyhedron using a bounding box, the object structure is simplified and the evaluation of collisions is significantly faster. This method allows simple collision-free situations to be resolved quickly.

3.2 Bottlenecks of Risk-based RRT*

We now focus more specifically on risk-based RRT*. This algorithm starts by generating a safe altitude map based on terrain altitudes, aircraft dynamics and landing site locations. This operation is computationally intensive, but since the terrain, landing site locations, and aircraft dynamics are invariant, once the map is generated, there is no need to generate it again. Thus, this map can be precomputed and its computation can be eliminated from the algorithm.

Next are the standard RRT* steps - sample, nearest neighbor search, steer, and configuration adding. From the previous section, we know that the sample and configuration adding steps have

constant time complexity. The steer step could be expensive if the search space contains a large number of obstacles. Since we neglect avoiding other aircraft here, the only requirement is to maintain a minimum safe altitude. We can therefore assume that the computational complexity of this step will be negligible. As a last step, there is a nearest neighbor search. We know that this operation can be computationally demanding if a complex formula is used to calculate the distance, as is the case here. In this algorithm, two relations are used to calculate the distance - maneuver length and risk. If the maneuver is represented by set of points connected by a line, then its length is calculated as the sum of the distances between these points. The computational complexity of this operation will therefore depend linearly on the number of points we use to represent the trajectory. However, even this operation is negligible compared to the risk calculation.

To calculate the risk, we use the relations (2.3) and (2.5-2.9). If we would implement the calculation according to these equations, we would have to evaluate the equations (2.5-2.9) at each point in the relevant area of space $\mathbb{R}^2$. Fortunately, we can modify the equations to reduce the calculation complexity. First, we substitute from the equations (2.5), (2.6) and (2.8) into the equation (2.3) to obtain

$$\mathcal{M}(q) = \int_{\mathbb{R}^2} p_{\text{imp}}(\mathbf{x}|\Gamma_{\text{bal}})\rho(\mathbf{x})A_{\text{exp}} \cdot \frac{1-k}{1-2k+\sqrt{\frac{\alpha}{\beta}}\left(\frac{\beta}{E}\right)^{\frac{3}{S(\mathbf{x})}}}\mathrm{d}\mathbf{x}. \tag{3.1}$$

We now define the expected shelter factor, as

$$\widetilde{S} = \int_{\mathbb{R}^2} p_{\text{imp}}(\mathbf{x}|\Gamma_{\text{bal}})S(\mathbf{x})\mathrm{d}\mathbf{x}, \tag{3.2}$$

which represents the average shelter factor in the impact area. If we substitute $S(\mathbf{x})$ for $\widetilde{S}$ in the equation (3.1), we can put the whole fraction with A_{exp} in front of the integral to obtain

$$\mathcal{M}(q) = A_{\text{exp}}\frac{1-k}{1-2k+\sqrt{\frac{\alpha}{\beta}}\left(\frac{\beta}{E}\right)^{\frac{3}{\widetilde{S}}}} \cdot I, \tag{3.3}$$

where

$$I = \int_{\mathbb{R}^2} p_{\text{imp}}(\mathbf{x}|\Gamma_{\text{bal}})\rho(\mathbf{x})\mathrm{d}\mathbf{x} \tag{3.4}$$

is the impact factor.

Now we need to modify the equations to make the calculation realizable on the CPU. To implement the computation of the area integral on the computer, we need to discretize the operation by converting the integral to a double summation. We can assume that at a sufficient distance from the position of the aircraft, the probability of collision p_{imp} is almost zero. This allows us to restrict the region over which the integral is computed and modify the relations (3.2) and (3.4) to

$$\widetilde{S} = \int_{x_t-\frac{a}{2}}^{x_t+\frac{a}{2}} \int_{y_t-\frac{a}{2}}^{y_t+\frac{a}{2}} p_{\text{imp}}((x,y)|\Gamma_{\text{bal}})S((x,y))\mathrm{dydx}, \tag{3.5}$$

$$I = \int_{x_t-\frac{a}{2}}^{x_t+\frac{a}{2}} \int_{y_t-\frac{a}{2}}^{y_t+\frac{a}{2}} p_{\text{imp}}((x,y)|\Gamma_{\text{bal}})\rho((x,y))\mathrm{dydx}, \tag{3.6}$$

where (x_t, y_t) are the coordinates of the center of the probability distribution and a is the length of the edge of the square in which the probability of impact is non-negligible. Now, since the region is finite, we can replace the integrals by a double summation,

$$\widetilde{S} = \sum_{i=-\frac{n}{2}}^{\frac{n}{2}} \sum_{j=-\frac{n}{2}}^{\frac{n}{2}} p_{\text{imp}}((x_t,y_t)+\frac{a}{n}(i,j))|\Gamma_{\text{bal}})S((x_t,y_t)+\frac{a}{n}(i,j)), \tag{3.7}$$

$$I = \sum_{i=-\frac{n}{2}}^{\frac{n}{2}} \sum_{j=-\frac{n}{2}}^{\frac{n}{2}} p_{\text{imp}}((x_t, y_t) + \frac{a}{n}(i,j))|\Gamma_{\text{bal}})\rho((x_t, y_t) + \frac{a}{n}(i,j)), \tag{3.8}$$

where n is the number of samples of the region along the x and y axes.

We see that $2n^2$ sums and products are needed to calculate the configuration risk, but this is less than evaluating the equations (2.5-2.9) at each point of the impact area. To achieve sufficient accuracy of the calculation, we will consider a value of n of at least 10. Thus, we can expect at least 200 sums and products. Moreover, to calculate the risk of the whole trajectory, we need to perform these 400 operations at several points that sufficiently cover the trajectory.

To calculate the risk, we also need to know the probability distribution of the impact p_{imp}. This can be determined by simulating the aircraft falling along a ballistic curve, which is given by the equation (2.4). Simulating an aircraft crash, moreover at several points along the trajectory, is a very demanding operation. However, since the probability of impact depends only on heading angle and height above the terrain, we can precompute the probabilities for several combinations of these parameters. Then, in the risk calculation itself, we select a matrix of probabilities corresponding to these parameters.

To speed up this algorithm, we should therefore focus primarily on the risk calculation, which involves the largest number of operations compared to the other parts of the algorithm.

Chapter 4
Proposed Method

In this section we first verify the theoretical results presented in the previous section with experimental benchmarking using an existing implementation in Julia [2]. We compare these results with the considerations presented in the previous section. To verify the speed of the algorithm on processors used for embedded devices, we rewrite the slowest part of the code in C++ and benchmark its performance.

Next we determine which part of the code is suitable for hardware acceleration. For this part of the algorithm we design a new component for the FPGA and verify its functionality. Next, we will measure the performance of the designed component and compare it to the software implementation.

4.1 Benchmark

In the Section 3.2 we evaluate the risk calculation as the most computationally demanding part of the algorithm. We see that the main problem is the calculation of the expected shelter factor $\widetilde{S}$ and the impact factor I. We verify this reasoning by benchmarking the existing Julia implementation [2]. The measurements were executed on an Intel Core i5-750 CPU with a frequency of 2.67 GHz. Table 4.1 contains the measured values.

From Table 4.1 we can see that the risk calculation takes in total $90\,\%$ of the time of the tree expansion. These values are therefore consistent with our conclusions from Section 3.2. However, when we look more closely at the risk calculation in the measurements, we find that $75\,\%$ of the time is taken up by finding the impact point where the risk is calculated. The actual calculations of the expected shelter factor and the impact factor take only $17\,\%$ of the time as seen in Table 4.2.

It would seem that the measurement results are inconsistent with the considerations in Section 3.2. This discrepancy can be justified by the fact that the measurements were performed on a desktop computer that contains a processor with a different architecture than the processors commonly used for embedded devices. An important difference here may be the size of the cache memory, the pipeline structure or the use of vector instructions. Therefore, this benchmark should be considered as indicative only.

In order to measure the complexity of the individual operations in the risk calculation correctly, we repeat this measurement on the Terasic DE10-Nano kit. To run the algorithm on HPS of this board, we need to rewrite the original Julia code in C++. Since the risk calculation accounts for $90\,\%$ of the computation time, we decided to implement only this part of the algorithm. This part includes finding the impact location by stepping along the precomputed ballistic curve, EWMM implementing the calculation of the expected shelter factor and impact factor, and the actual risk calculation according to the equations (3.7) and (3.8). The necessary input data, including the precomputed ballistic curve, impact probability matrices, shelter map and population density map, were generated using the Julia implementation. We saved this data in files from which it is loaded when run on the kit which we consider a valid approach for this proof-of-concept work.

Table 4.1: Benchmark of Julia implementation.

Function	No. of calls [-]	Total time [s]	Perc. of time [%]	Single call time [ms]
RRT* expansion	1	159.00	99.5 %	159 000.00
└ RrtStar add sample	588	158.00	98.9 %	269.00
└ Add node	418	158.00	98.7 %	377.00
└ Find best parent	418	79.20	49.6 %	189.00
└ Counting maneuver risk	3690	77.60	48.6 %	21.00
└ **Get risk at q**	**296000**	**73.20**	**45.8 %**	**0.25**
└ Coordinates check	297000	1.82	1.1 %	0.01
└ Collision check	297000	0.70	0.4 %	<0.01
└ ENU to MAP	297000	0.66	0.4 %	<0.01
└ Create maneuver	3690	1.02	0.6 %	0.28
└ Rewire	377	64.10	40.2 %	170.00
└ Counting maneuver risk	4150	62.60	39.2 %	15.10
└ **Get risk at q**	**254000**	**58.90**	**36.9 %**	**0.23**
└ Coordinates check	254000	1.56	1.0 %	0.01
└ Collision check	254000	0.59	0.4 %	<0.01
└ ENU to MAP	254000	0.56	0.4 %	<0.01
└ KNN	377	0.18	0.1 %	0.47
└ Counting maneuver risk	381	7.90	4.9 %	20.70
└ **Get risk at q**	**32300**	**7.43**	**4.7 %**	**0.23**
└ Coordinates check	32300	0.20	0.1 %	0.01
└ Collision check	32300	0.08	0.0 %	<0.01
└ ENU to MAP	32300	0.07	0.0 %	<0.01
└ Steer	381	4.37	2.7 %	11.50
└ Counting maneuver risk	304	4.29	2.7 %	14.10
└ **Get risk at q**	**15700**	**4.06**	**2.5 %**	**0.26**
└ Coordinates check	15700	0.10	0.1 %	0.01
└ Collision check	15700	0.04	0.0 %	<0.01
└ ENU to MAP	15700	0.03	0.0 %	<0.01
└ KNN	795	0.50	0.3 %	0.63
└ Get near nodes	418	0.13	0.1 %	0.30
└ Rewire	2	0.21	0.1 %	107.00
└ Find best parent	2	0.21	0.1 %	107.00
└ Counting maneuver risk	2	0.01	0.0 %	6.78
└ Steer	2	0.01	0.0 %	3.66
└ KNN	4	0.01	0.0 %	0.53
└ Get near nodes	2	0.00	0.0 %	0.17
RrtStar Init	1	0.69	0.4 %	687.00
Counting maneuver risk	9	0.14	0.1 %	15.30

Table 4.2: Benchmark of risk computing in Julia.

Function	No. of calls [-]$\times 10^3$	Total time [s]	Perc. of time [%]	Single call time [µs]
Find crash location	602	105	75.2 %	174.0
└ Stepping on balistic curve	602	102	72.9 %	169.0
└ Single step	10 000	96.6	69.3 %	9.7
└ Coordinates check	10 000	39.3	28.2 %	3.9
└ Position computing	10 000	18.4	13.2 %	1.8
└ Collision check	10 000	13.7	9.8 %	1.4
└ ENU to MAP	10 000	9.54	6.8 %	1.0
└ Crash check	10 000	2.78	2.0 %	0.3
└ Get fall	602	0.3	0.2 %	0.4
Compute risk	602	23.6	16.9 %	39.2
Get impact property	602	10.9	7.8 %	18.0

Table 4.3: Benchmark of risk computing on DE10-Nano.

Function	Average time [µs]
Get risk	38.58
└ Find crash location	17.17
└ Variable init	1.26
└ Stepping on balistic curve	15.22
└ ENU to MAP	0.69
└ Get impact property	2.72
└ Compute risk	18.69

Once we have the test code and data ready, we can take a second measurement. The measurement was repeated 100 times and the average was calculated from the measured times. The results are listed in Table 4.3.

We see that when run on the kit, the EWMM calculation has a significantly larger impact. Despite this, finding the impact position has almost 40 % impact on performance. The impact position search is implemented as a for loop in which the aircraft is moved along each point of the precomputed ballistic curve. At each iteration, a collision with the minimum safe altitude level must be detected. It is also necessary to check that the ballistic curve does not deviate from the boundaries of the maps stored in memory. Since the loop is executed on average 17 times, this operation also has a large impact on performance.

As a first proposed speed-up we use the interval halving method to find the impact position. Since in our case the ballistic curve is precomputed at 42 points, only 6 repetitions are needed to find the impact position instead of the average 17. The values are summarized in Table 4.4 and represent the average of 100 measurements.

We see that after software optimization using the interval halving method, the impact position search time dropped to 26 %. This also relatively increases the calculation time of EWMM to 59 % of the risk calculation time. Since EWMM contains a large number of operations, we cannot speed up this calculation in software significantly.

Table 4.4: Benchmark of risk computing on DE10-Nano after loop optimization.

Function	**Average time** [µs]
Get risk	32.29
└ Find crash location	10.61
└ Variable init	1.41
└ Stepping on balistic curve	8.53
└ ENU to MAP	0.67
└ Get impact property	2.67
└ Compute risk	19.01

4.2 Component Design

There are two computationally intensive operations identified in the previous section. The first operation is EWMM used to calculate the expected shelter factor and the impact factor. This operation forms 59 % of the configuration risk calculation. The other demanding operation is to find the impact location, which despite optimization using the interval halving method still constitutes 33 % of the configuration risk calculation time. Now we can focus on optimizing the calculation using FPGA.

While the structure of the CPU is designed for fast processing of a series of instructions on which the computation runs very fast, we have very limited options for computing parallelizable tasks. For these tasks it is convenient to use FPGA, which consists of a large number of small universal blocks that can be arbitrarily connected as described in Section 2.2. This allows us to create a circuit specifically designed for parallel processing.

Finding the point of impact is a difficult task to parallelize and therefore we would not gain much by implementing this part on FPGA. Another option would be to consider stream processing with pipelining, however, this is out of the scope of this thesis, as it would require us to completely change the principle of the algorithm. Computation of EWMM can be fully parallelized and thus achieve a significant speedup over the CPU implementation.

Our goal, then, is to implement a component to compute Element-wise Matrix Multiplication, which we will use to compute the expected shelter factor and the impact factor. In these calculations, one matrix contains the population density and shelter maps, which are invariant, respectively, while the other matrix contains the impact probability, which is dependent on the position and speed of the aircraft and will need to be dynamically changed.

Since in both cases one matrix is immutable, it would be useful to store this data in memory on FPGA, thus reducing the amount of data transferred. However, this is not possible since the memory built into FPGA is not large enough. Therefore, we will design the component so that instead of sending the entire maps to FPGA, we only send the parts that are used in a given computation. However, these parts will have to be changed depending on the position of the impact probability matrix.

To test the calculation on FPGA we chose the following architecture. The circuit interface consists of two memories of 50×50 numbers of type double, used to store the matrices, and a memory of 20 bytes, used to pass the sizes of the matrices and return the computed result. Since using floating point arithmetic would unnecessarily complexify the circuit, we use a fixed point data type to represent the numbers. For this purpose, we used a signless representation with 64 bits of the decimal part and 16 bits of the integer part. The numbers at the corresponding positions in memory are first converted from the double type to our fixed point type. Then the converted numbers are multiplied and the result is added to the accumulator. After EWMM is completed, the result is returned as a fixed point number. The conversion back to double type is left to the CPU for efficiency.

For the purpose of easy modifiability and speed of design, we implement the component using

HLS. We marked the component as `hls_avalon_slave_component`, which adds start, busy and done signals. The important signal for us is the done signal, which lets us know when the calculation is finished and we can read the result from the component. To define memory to store matrices and parameters, we used `hls_avalon_slave_memory_argument`, which creates a memory mapped slave interface. This allows us to communicate directly with the CPU. In the passed parameters, we specify the sizes of the matrices, so that in case of smaller matrices, we can terminate the calculation correctly and not add false values. To store the sum, we define a fixed point register. This is followed by a loop defining the calculation itself. The loop is marked with `#pragma ivdep` to tell the compiler that there are no data dependencies between loop cycles. This allows a more efficient circuit to be generated. Finally, we split the 80 bit result in fixed point representation into 32 bit parts to store the result in parameter memory. This result will be read on the CPU and converted to double type. The whole component description in HLS is listed in algorithm 3.

Algorithm 3 HLS code of component

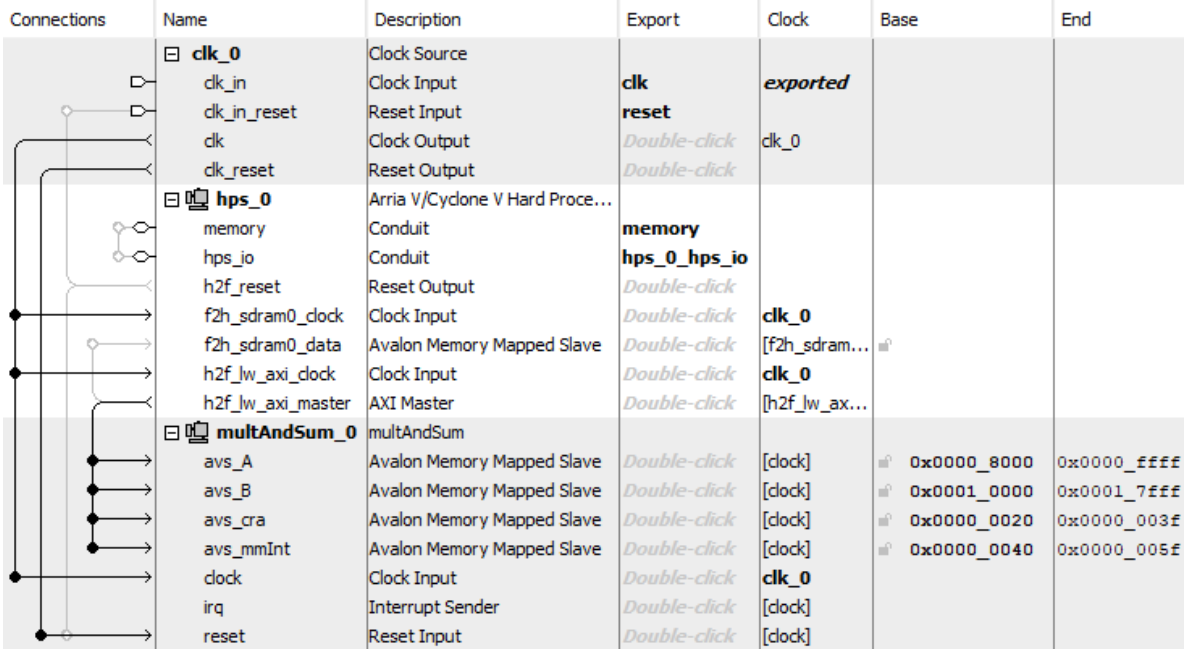
```
component hls_avalon_slave_component void multAndSum(
    hls_avalon_slave_memory_argument(
                5*sizeof(uint32_t)) uint32_t* mmInt,
    hls_avalon_slave_memory_argument(
                N*N*sizeof(double)) double* A,
    hls_avalon_slave_memory_argument(
                N*N*sizeof(double)) double* B) {
    uint32_t width = mmInt[0];
    uint32_t height = mmInt[1];

    fixed sum = 0;
    #pragma ivdep
    for(uint32_t i = 0; i < N * N; i++) {
        fixed fa, fb;
        fa = static_cast<fixed>(A[i]);
        fb = static_cast<fixed>(B[i]);
        sum += fa * fb;

        if(i >= width * height) break;
    }

    mmInt[2] = sum.slc<16>(64);
    mmInt[3] = sum.slc<32>(32);
    mmInt[4] = sum.slc<32>(0);
}
```

The designed component is compiled into VHDL and connected to the system on the DE10-nano kit in the Platform designer in Quartus. The inputs of the component were connected to the AXI bus. The new input `avs_cra` contains the signals defined by `hls_avalon_slave_component`. The whole SoPC architecture is shown in Figure 4.1.



Connections	Name	Description	Export	Clock	Base	End
	clk_0	Clock Source				
	clk_in	Clock Input	**clk**	***exported***		
	clk_in_reset	Reset Input	**reset**			
	clk	Clock Output	*Double-click*	clk_0		
	clk_reset	Reset Output	*Double-click*			
	hps_0	Arria V/Cyclone V Hard Proce...				
	memory	Conduit	**memory**			
	hps_io	Conduit	**hps_0_hps_io**			
	h2f_reset	Reset Output	*Double-click*			
	f2h_sdram0_clock	Clock Input	*Double-click*	**clk_0**		
	f2h_sdram0_data	Avalon Memory Mapped Slave	*Double-click*	[f2h_sdram...		
	h2f_lw_axi_clock	Clock Input	*Double-click*	**clk_0**		
	h2f_lw_axi_master	AXI Master	*Double-click*	[h2f_lw_ax...		
	multAndSum_0	multAndSum				
	avs_A	Avalon Memory Mapped Slave	*Double-click*	[clock]	0x0000_8000	0x0000_ffff
	avs_B	Avalon Memory Mapped Slave	*Double-click*	[clock]	0x0001_0000	0x0001_7fff
	avs_cra	Avalon Memory Mapped Slave	*Double-click*	[clock]	0x0000_0020	0x0000_003f
	avs_mmInt	Avalon Memory Mapped Slave	*Double-click*	[clock]	0x0000_0040	0x0000_005f
	clock	Clock Input	*Double-click*	**clk_0**		
	irq	Interrupt Sender	*Double-click*	[clock]		
	reset	Reset Input	*Double-click*	[clock]		

Figure 4.1: Connection of a component in Platform designer.

4.3 Measuring Hardware Acceleration

Now when the component has been created, we can test its functionality and properties. After adding the component to the test program, we encountered a problem with incorrect results. It turns out that when the calculation is repeated, the following results are affected by the previous data. This error should have been prevented by a condition checking the sizes of the matrix, but the compiler probably removed this condition as part of the circuit optimization. When we zeroed-out the memory before loading the new data, the result was already correct.

Now when the component is working correctly, we measure the speed of the calculation. We took the measurements a hundred times and the averaged results are reported in Table 4.5.

We can see that zeroing-out the memory now takes most of the time, which makes the computation significantly slower. Even if we solve this problem directly within the component and thus eliminate the need for memory zeroing, the calculation would still slow down the data transfer. However, we can notice that after the data transfer is complete, the wait time for the computation to complete is less than 15 µs. We therefore focused on measuring the latency of the circuit. We measure it as a time from the change of a random element in the matrix to the execution of a new computation signaled by a done signal. For this component, we measured a latency of only 20 µs. This means that if we eliminate the need of zeroing-out the memory and minimize the amount of data transferred, we can achieve the same computational performance on FPGA, which in our case is due to the data transfers 16 times slower than the CPU. Assuming we place this component multiple times in parallel on FPGA which should be possible, as the current design occupies only 36 % of the FPGA fabric, we would achieve the desired speedup of the whole algorithm. In the next section, we therefore focus on discussion of the ways to minimize the amount of data needed for the computation and to speed up the data transfer required for the computation.

Table 4.5: Benchmark of risk computing using FPGA component.

Function	**Average time** [µs]
Get risk	4639.194
└ Find crash location	11.51
└ Variable init	1.54
└ Stepping on balistic curve	9.25
└ ENU to MAP	0.72
└ Get impact property	2.90
└ Compute risk	4624.78
└ Risk evaluation	4.19
└ Impact factor	2312.23
└ Zeroing	2111.47
└ Data transfer	180.34
└ Wait for valid result	14.97
└ Communication overhead	5.45
└ Shelter factor	2308.36
└ Zeroing	2109.60
└ Data transfer	178.55
└ Wait for valid result	14.94
└ Communication overhead	5.26

Chapter 5
Discussion of the Results

We identified the slowest part of the algorithm using the Julia implementation and rewrote this part in C++ so that we could run it on a processor whose architecture is similar to conventional embedded processors. This allowed us to obtain accurate performance measurements. Then, for the slowest part of the algorithm, we designed and tested the component on FPGA in order to speed up the computation. Although we did not achieve the overall speedup with this component, we found that it has low latency. In this section, we therefore focus on the possibilities of modifying the architecture to minimize the amount of data transferred and thus utilize the low latency of the circuit. We first present several optimization possibilities that we will eventually use in the design of the final architecture.

First and foremost, we should solve the problem of the need for zeroing-out, which causes the greatest computational bottleneck. Since the memory on FPGA is implemented by a memory block, it is not possible to delete all the data at once. Therefore, it will not be possible to implement zeroing within the component. The simplest solution is to modify the software part to make all generated impact probability matrices have the same size. This will ensure that all data in memory will always be overwritten and there will be no need to zeroing-out any data.

However, this modification is likely to increase the amount of data that needs to be transferred to FPGA. This brings us to the second problem, which is data transfer, which even without the need for zeroing-out slows down the computation significantly. To reduce the amount of data, we can also implement the calculation of the impact probability matrix on FPGA. Instead of the matrices themselves, the ballistic fall parameters from which the matrix will be generated will be passed to FPGA. In our opinion, calculating the ballistic fall convolution matrix from the parameters is straightforward. This modification will reduce the amount of data transferred by almost half.

We also know that the risk of a trajectory is calculated as the integral of the configuration risks along this trajectory (2.2). In practice, the integral is realized as the sum of the configuration risks at many points along the trajectory. Hence, when calculating the trajectory risk, the individual configuration risks are calculated over the same map area. This means that individual map cuts overlap when calculating trajectory risk. We verified this deduction using the Julia implementation, where we measured an overlap rate of more than $90\,\%$ in most cases. If we merge the risk calculation of adjacent configurations, we could significantly reduce the amount of map data transferred to FPGA. The idea is visually shown in Figure 5.1.

We can further improve the efficiency of the data transfer by using Direct Memory Access (DMA). By delegating the data transfer to the DMA controller, we allow the CPU to perform other operations in parallel with the data transfer. However, transfering each map slice using DMA would require a lot of repetitive configurations of the DMA transfer process. Therefore, it seems useful to merge the overlapping map slices, even at the cost of transferring unused data.

The modifications we described above, especially the calculation of the FPGA impact probability matrices, still require a large number of memory blocks to store the data. It would be useful to implement the component using as little memory as possible so that we could use this free space for multiple instances of the component in parallel or for other circuits. For this purpose, we can switch to a streaming architecture. Instead of a memory-mapped slave interface, we can use a stream input that uses less memory. Once it gets an element of the map matrix on the input, it immediately multiplies it with the corresponding element in the probability matrix and adds the result to the accumulator. Since these matrix elements are no longer needed, they can be discarded. The only required memory in such

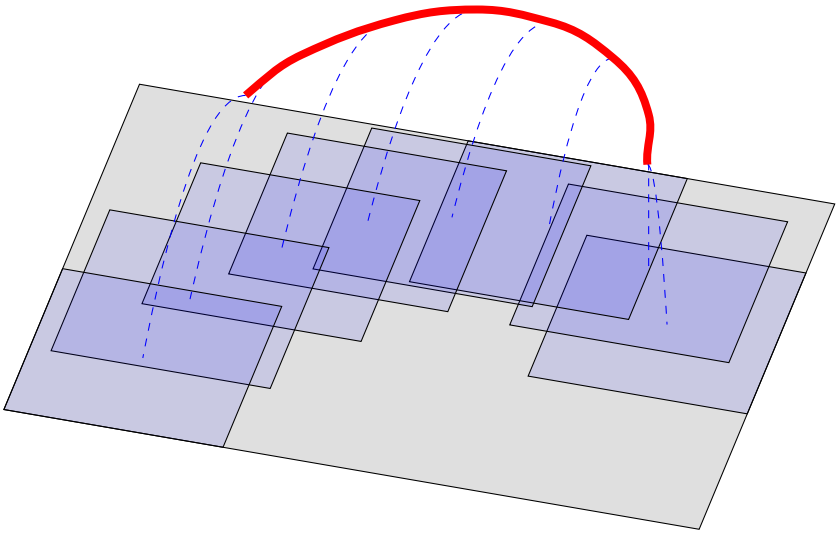


Figure 5.1: Calculation of risk along the trajectory. Using a single map cut to calculate multiple risk configurations along a trajectory reduces the amount of data transferred to the FPGA.

a design is the one for the intermittent calculations and the accumulator register. If it were possible to modify the generation of the impact probability matrix in the same way, we would reduce the memory requirements significantly.

In our future work we will apply these ideas and design an architecture that could already achieve the desired speedup. The whole architecture will be designed as streaming, so we will not be limited by the amount of memory as in the previous case which will allow us to increase the parallelization. The interface of this component will include a memory-mapped slave interface for returning results and for parameterizing multiple ballistic falls, and a stream input for map data. After setting the ballistic falls parameters and running DMA to transfer the map data, the elements of the impact probability matrices will be generated in parallel from the ballistic fall parameters. The generated impact probability data will be streamed to the multipliers using the streaming bus along with the map data. The results of the multiplication will be added to the accumulators. Due to the parallel computation of the probability matrices, we can use one map cut multiple times. The result will be the simultaneous output of several risk configurations, which will be written from the accumulators to the memory-mapped slave interface and read by the CPU. The schema of this architecture is shown in Figure 5.2.

This architecture will certainly have a higher latency, than the previous architecture. On the other hand, when the computation is completed, we get several computed risks simultaneously. In addition, by distributing the work between the component and the DMA controller, the processor will be able to continue computing the algorithm and prepare the data for the calculation of another risks.

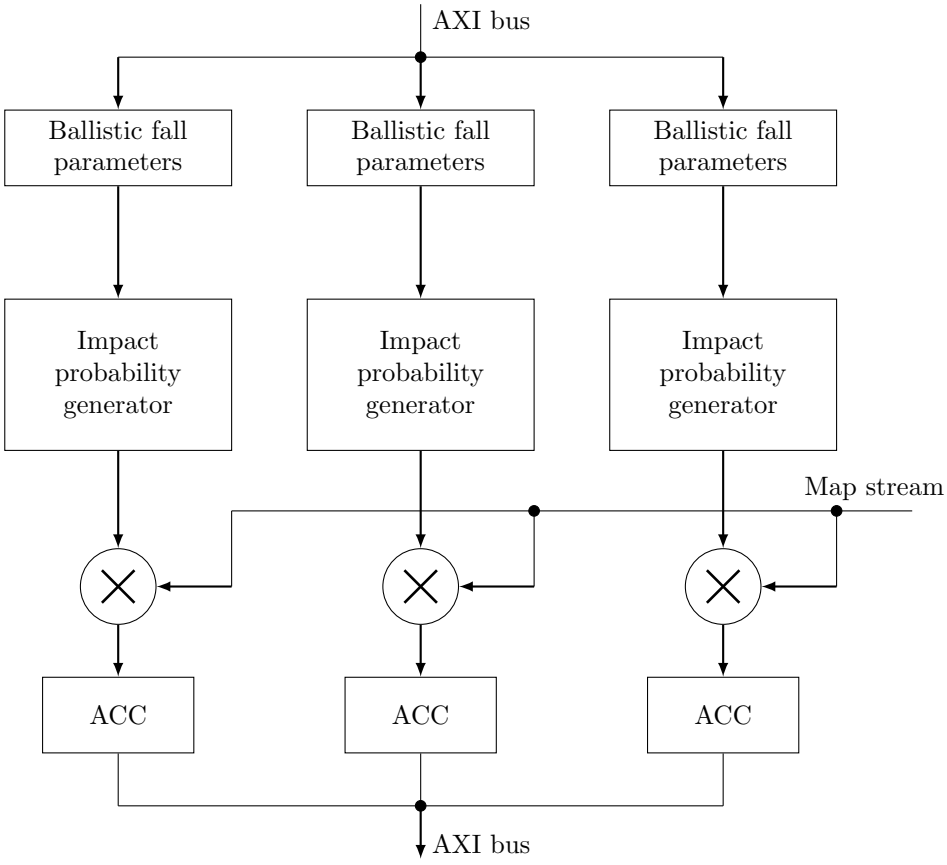


Figure 5.2: Streaming architecture. The elements of the impact probability matrix are generated from the given ballistic fall parameters. These are multiplied with the corresponding elements of the map cut and added to the accumulator (ACC). The result is read from the accumulator by the CPU.

Chapter 6
Conclusion

In this work, we focused on hardware acceleration of the Risk-based RRT* algorithm used to find the trajectory of the aircraft that minimizes the amount of damage caused by a potential crash. First, we theoretically examined the computational complexity of the individual parts of the RRT-based algorithms, and in more detail of the Risk-based RRT* algorithm. In the case of the Risk-based RRT*, we evaluated the risk calculation as the most demanding operation, due to the frequent computation of EWMM.

To verify this theoretical result, we benchmarked the existing Julia implementation. The measurement results exactly confirmed our hypothesis. A more detailed benchmarking of the risk calculation showed that the most challenging operation in the risk calculation is the search for the impact location, not the calculation of EWMM as we had assumed. However, this discrepancy was caused by the specific architecture of the desktop CPUs on which the measurement was performed.

For this reason, we rewrote the risk calculation in C++. This allowed us to perform the same measurement on a CPU whose architecture is similar to CPUs used for embedded devices. After optimizing this part of the algorithm, we already got results that matched the theoretical reasoning.

Based on these results, we decided to accelerate the computation using custom designed FPGA architecture. We designed a component to compute EWMM using HLS and connected it to the system on the used embedded development board. This component is a proof-of-concept work that helped us understand and benchmark the performance of the hardware-accelerated algorithm. We found, that the latency of the designed component is approximately the same as the computation time on the CPU. Thus, we focused on the possibilities of reducing the amount of data transferred, which represented the main computational slowdown. Furthermore, we tried to optimize the memory usage within FPGA to provide more parallelization capabilities. Finally, using these improvements, we designed a new component based on a streaming architecture that could achieve computational speedup. Thus, we believe that all the points of the thesis assignment have been met.

In future work, it would be desirable to verify the functionality of the proposed architecture and implement the whole algorithm in C++ to obtain accurate measurements of the achieved speedup.

References

[1] Stefano Primatesta, Matteo Scanavino, Giorgio Guglieri, and Alessandro Rizzo. A risk-based path planning strategy to compute optimum risk path for unmanned aircraft systems over populated areas. In *International Conference on Unmanned Aircraft Systems (ICUAS)*, pages 641–650, 2020.

[2] Jakub Sláma, Petr Váňa, and Jan Faigl. Risk-aware trajectory planning in urban environments with safe emergency landing guarantee. In *IEEE 17th International Conference on Automation Science and Engineering (CASE)*, pages 1606–1612, 2021.

[3] Jonathan D. Gammell, Siddhartha S. Srinivasa, and Timothy D. Barfoot. Informed RRT*: Optimal sampling-based path planning focused via direct sampling of an admissible ellipsoidal heuristic. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2997–3004, 2014.

[4] Fahad Islam, Jauwairia Nasir, Usman Malik, Yasar Ayaz, and Osman Hasan. RRT*-smart: Rapid convergence implementation of RRT* towards optimal solution. In *IEEE International Conference on Mechatronics and Automation*, pages 1651–1656, 2012.

[5] Ma Han, Meng Fei, Ye Chengwei, Wang Jiankun, and Max Qinghu Meng. Bi-Risk-RRT based efficient motion planning for autonomous ground vehicles. *IEEE Transactions on Intelligent Vehicles*, 7(3):722–733, 2022.

[6] Steven M. LaValle. Rapidly-exploring random trees: A new tool for path planning. *Computer Science Dept. Oct.*, 98(11), 1998.

[7] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[8] S. M. LaValle. *Planning Algorithms*. Cambridge University Press, Cambridge, U.K., 2006. Available at http://planning.cs.uiuc.edu/.

[9] Sertac Karaman and Emilio Frazzoli. Optimal kinodynamic motion planning using incremental sampling-based methods. In *49th IEEE Conference on Decision and Control (CDC)*, pages 7681–7687, 2010.

[10] Ioan A. Şucan, Mark Moll, and Lydia E. Kavraki. The Open Motion Planning Library. *IEEE Robotics & Automation Magazine*, 19(4):72–82, 2012.

[11] Kourosh Naderi, Joose Rajamaki, and Perttu Hämäläinen. RT-RRT*: A real-time path planning algorithm based on RRT*. pages 113–118, 2015.

[12] Petr Váňa, Jakub Sláma, and Jan Faigl. Surveillance planning with safe emergency landing guarantee for fixed-wing aircraft. *Robotics and Autonomous Systems*, 133:103644, 2020.

[13] Hamidreza Chitsaz and Steven M. LaValle. Time-optimal paths for a dubins airplane. In *46th IEEE Conference on Decision and Control (CDC)*, pages 2379–2384, 2007.

[14] Vaughn Betz and Jonathan Rose. How much logic should go in an FPGA logic block. *IEEE Design & Test of Computers*, 15(1):10–15, 1998.

[15] King Lum Cheung and Ada Wai-Chee Fu. Enhanced nearest neighbour search on the R-Tree. *ACM SIGMOD Record*, 27(3):16–21, 1998.

[16] S. Gottschalk, M. C. Lin, and D. Manocha. OBBTree: A hierarchical structure for rapid interference detection. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, page 171–180, 1996.