École Polytechnique Fédérale de Lausanne
Dartmouth College

Hardware-Algorithm Co-Design for Real-Time Linear Model Predictive Control

FPGA Implementation and Deployment on a Resource-Constrained Quadrotor

Andrea Grillo

Master's Thesis

Prof. Brian Plancher
Supervisor, Dartmouth College – A2R Lab

Prof. Colin Neil Jones
Supervisor, EPFL – Automatic Control Lab

March 13, 2026

Acknowledgments

I would like to thank everyone who contributed to making this thesis possible.
I am especially grateful to Brian Plancher for hosting me at Dartmouth and for his endless energy and enthusiasm, as well as for being the most available advisor I could have asked for. Sometimes to the point where I wondered if he ever actually sleeps.
I would like to thank my advisor, Colin Jones, for his support and guidance during my journey at EPFL.
I thank the entire A2R team for making it such a welcoming and stimulating place to work. A special thank you to Roy Xing, my lab buddy, for the many insightful conversations and for helping during the experiments. I really enjoyed working alongside you in the lab. I also thank Gabriel Bravo for his expert advice and for the many lunches together, and bringing a touch of Latin American spirit to the lab. Thanks to Ishaan Mahajan for the useful insights into TinyMPC, and to Jianghan Zhang for contributing to the great lab environment.
Thanks also to other members of the lab: Yewei Huang for the delicious food and crazy stories, Amel Docena for his friendship and for helping me feel at home in Hanover, and Yitao Jiang, who always seems to be working on the coolest and craziest hardware projects.
Thanks to Ashish Rao Mangalore for all the insights and information about the Intel Loihi 2 platform.
I am also very grateful to Roland Schwan and Gösta Stomberg, with whom I had the chance to work on my first research project. It was a challenging and deeply instructive experience that played an important role in setting me on this path.
A big thank you to Shaohui Yang for introducing me to Brian and the A2R Lab and making this whole experience possible.
I am also grateful to Jacob Willis for taking the time to review the PCB design.
A big thank you to my friends and classmates at EPFL – Chiara Evangelisti, Elisa Ferrara, Riccardo Lionetto, Francesco Maglie, Angelo Giovine, Giada Ehrlich, Enrico Pinelli, and Allegra Magnetti – with whom I have shared many intense and unforgettable moments at EPFL, those years would not have been the same without you.
A very special thank you to Giulia, whose support has helped me far more than she probably realizes, often in ways she cannot even imagine. And for everything still to come.
Thanks also to my brother Davide for his support and for patiently helping me with the video and image processing for this thesis. His perspective at the intersection of technical and artistic thinking is always invaluable.

Thank you to my parents, Luigi and Stella, for their constant support over the years, and for always standing behind me in everything I chose to pursue, whether they agreed or not. Their advice has always been there whenever I need it.
Finally, I would like to thank my grandparents for their affection and for sharing with me the wisdom and perspective of a lifetime.

Hanover, March 13, 2026 Andrea Grillo

Abstract

Model Predictive Control (MPC) is a powerful framework for feedback control, enabling controllers that explicitly account for system dynamics and operational constraints. However, the need to repeatedly solve a constrained optimization problem online makes MPC computationally expensive and energy-intensive, posing a fundamental challenge for deployment on resource-constrained robotic platforms. For small embedded robots such as nano-quadrotors, this challenge is acute. Recent work has demonstrated that carefully formulated linear MPC can run onboard microcontrollers, but such platforms face hard limits in computational throughput and energy budget. As problem complexity grows with longer prediction horizons, tighter iteration budgets, or stricter constraints, microcontroller-based implementations become impractical.

Custom computing hardware offers a path forward. By tailoring computation to the structure of a specific algorithm, domain-specific hardware can achieve substantial improvements in both throughput and energy efficiency compared to general-purpose processors. Field-Programmable Gate Arrays (FPGAs) provide a flexible platform for prototyping such custom architectures, while Application-Specific Integrated Circuits (ASICs) can deliver even greater efficiency in production systems. Emerging neuromorphic architectures offer a complementary direction, with the potential for ultra-low-power operation through fundamentally different computational principles.

This thesis investigates how such alternative architectures can extend the capability of optimization-based control on constrained robots, with demonstrations on a nano-quadrotor running linear MPC. The primary contribution is a hardware-accelerated linear MPC solver based on a custom FPGA datapath implementation of the Alternating Direction Method of Multipliers (ADMM) algorithm, co-designed for reduced-precision arithmetic. We also develop a custom FPGA expansion board for the Crazyflie platform. Through these advances, we demonstrate fully onboard constrained MPC at high control frequencies, achieving approximately 15× lower solver latency and 200× improvement in energy–delay product relative to state-of-the-art baselines. This enables longer horizons, more solver iterations per control step, and more reliable constraint handling. In parallel, a preliminary feasibility analysis of mapping iterative optimization algorithms onto Intel's Loihi 2 neuromorphic processor is explored as a promising direction toward ultra-low-power implementations in future work. Together, these contributions illustrate how algorithm-hardware co-design can significantly extend the practicality of optimization-based control on resource-constrained robotic platforms.

Contents

Acknowledgments **2**

Abstract **4**

1 Introduction **8**
1.1 Motivation and Problem Statement 8
1.2 Hardware-Aware MPC for Embedded Systems 10
1.3 Contributions and Thesis Organization 11
1.3.1 Code and Hardware Repositories 12

2 Background **14**
2.1 Model Predictive Control 14
2.1.1 System Modeling for MPC 14
2.1.2 Finite-Horizon Optimal Control 15
2.2 Quadratic Programs in MPC 16
2.2.1 Optimality Conditions 16
2.3 Optimization Algorithms for Embedded MPC 17
2.3.1 Active-Set Methods 17
2.3.2 Interior-Point Methods 17
2.3.3 First-Order Methods 18
2.4 Real-Time Optimization Challenges 18
2.5 Hardware Platforms for Real-Time Optimization 19
2.5.1 Microcontrollers and CPUs 19
2.5.2 Graphics Processing Units 20
2.5.3 Field-Programmable Gate Arrays 20
2.5.4 Emerging Computing Platforms 20

3 System Modeling and MPC Problem Formulation **22**
3.1 Quadrotor Control as a Benchmark Problem 22
3.2 Nonlinear Quadrotor Dynamics 23
3.2.1 Translational Dynamics 23
3.2.2 Rotational Dynamics 24

3.3 Linearization Around Hover . . . 25
3.4 Linear MPC Formulation . . . 25
3.4.1 Quadratic Program Formulation . . . 26
3.4.2 Inequality Constraints . . . 27
3.5 Structure of the Resulting Optimization Problem . . . 27
3.6 The ADMM Algorithm . . . 28
3.6.1 ADMM Formulation . . . 28
3.6.2 Augmented Lagrangian . . . 28
3.6.3 ADMM Iterations . . . 29
3.6.4 Convergence . . . 29
3.6.5 Application to Quadratic Programs . . . 29
3.7 Closed-Loop MPC Execution with ADMM . . . 31

4 Hardware–Algorithm Co-Design and FPGA Implementation 33
4.1 Motivation and Related Work . . . 33
4.2 Preliminary work . . . 35
4.2.1 FPGA Selection and Sizing Considerations . . . 35
4.2.2 High-Level Synthesis . . . 38
4.3 Algorithm Co-Design . . . 39
4.3.1 Linear System Solving . . . 41
4.3.2 Efficient Storage of Matrices . . . 43
4.3.3 Hardware-Oriented Implementation of the ADMM Solver . . . 44
4.3.4 Computational Complexity . . . 46
4.3.5 Floating-Point vs Fixed-Point Datatypes . . . 48
4.3.6 Warm Starting of the Solver . . . 48
4.3.7 Handling of the ρ Parameter . . . 49
4.3.8 Accumulator Data Width . . . 49
4.3.9 Setpoint Feeding . . . 49
4.4 FPGA Onboard: PCB Design . . . 50
4.4.1 PCB Layer Stackup . . . 52
4.4.2 Power Supply and Sequencing . . . 53
4.4.3 Connection to Crazyflie . . . 54
4.4.4 Clock . . . 54
4.4.5 Programming Interface . . . 54
4.4.6 SPI Flash . . . 54
4.4.7 LEDs . . . 54
4.5 System Integration . . . 55
4.5.1 Integrating the HLS IP in an FPGA Project . . . 55
4.5.2 Automated Build and Deployment Pipeline . . . 56
4.5.3 SPI Communication . . . 56
4.5.4 Crazyflie Firmware Integration . . . 57

4.5.5 Board Manufacturing and Drone Integration 58

5 Experimental Evaluation **60**
5.1 Solver Benchmarking 60
5.1.1 Latency and Timing Analysis 60
5.1.2 Resource Utilization Analysis 66
5.1.3 Energy Consumption and Energy-Delay Product 67
5.2 Onboard Flight Experiments 69
5.2.1 Experimental Setup 69
5.2.2 Unconstrained Figure-Eight Trajectory Tracking 71
5.2.3 Constrained Trajectory Tracking 73

6 Exploratory Study: Optimization on Neuromorphic Hardware **77**
6.1 Motivation and Scope 77
6.2 Overview of Loihi 2 78
6.2.1 Core Architecture 79
6.2.2 Data Representation on Loihi 2 79
6.2.3 Programmable Microcode 81
6.3 Neuromorphic Formulations of Quadratic Programming 81
6.4 Mapping Structured Optimization Algorithms 84
6.5 Quantization and Weight Representation 85
6.6 Preliminary Experiments 86
6.7 Discussion 87

7 Conclusion and Future Work **89**
7.1 Conclusion 89
7.2 Future Work 90
7.2.1 Engineering Improvements 90
7.2.2 Future Research Directions 91

Bibliography **95**

Chapter 1

Introduction

1.1 Motivation and Problem Statement

Small robotic platforms must execute feedback control loops at high frequencies while operating under extremely limited computational and energy budgets. These constraints arise across many resource-constrained robotic systems where size, weight, and power limitations severely restrict the available onboard computing resources. At the same time, these systems increasingly operate in dynamic environments where safety, performance, and agility require controllers that can explicitly account for system dynamics and operational constraints.

Model Predictive Control (MPC) provides a principled framework to address these challenges. MPC repeatedly solves a constrained optimization problem that predicts the future evolution of the system and computes control inputs that minimize a cost function while satisfying state and input constraints [1–4]. This capability makes MPC particularly attractive for robotic systems where safety constraints and dynamic performance are critical.

Despite these advantages, deploying MPC on embedded robotic platforms remains challenging. At each control step, linear MPC requires solving a constrained quadratic program (QP) whose computational complexity increases with the prediction horizon, system dimension, and number of constraints. Although convex QPs can be solved efficiently using well-established numerical methods [5, 6], executing these computations within the tight timing constraints of high-frequency control loops remains difficult on embedded processors with limited computational resources.

These challenges become particularly pronounced for nano-scale aerial vehicles such as the Crazyflie nano-quadrotor [7]. Platforms of this scale exhibit fast and often unstable dynamics that require control loops operating at very high frequencies, while their onboard microcontrollers provide only a fraction of the computational capability available on larger robotic systems. As a result, such systems represent an extreme test case for embedded control: if advanced optimization-based

controllers can be executed reliably on a platform weighing only tens of grams, similar approaches may be feasible across a broad class of resource-constrained robotic systems.

In practice, many advanced control strategies for nano-quadrotors therefore rely either on simplified control laws or on off-board computation [8–16], where the optimization problem is solved on an external computer and the resulting control inputs are transmitted to the robot. While this approach enables the use of sophisticated algorithms, it introduces communication dependencies and prevents the control system from operating fully onboard.

Achieving fully onboard MPC on such systems remains a significant challenge. In the specific case of the Crazyflie platform, TinyMPC [17] is currently the only solver specifically designed to enable real-time MPC within the computational constraints of the onboard microcontroller. Although this work demonstrates that MPC can in principle be executed on such platforms, the extremely limited computational resources constrain the number of solver iterations that can be performed within a high-frequency control loop. As a result, prediction horizons remain short and constraint handling capabilities are limited.

These limitations suggest that further progress cannot rely solely on improved software implementations on general-purpose embedded processors. Instead, exploiting specialized computing architectures that accelerate the optimization algorithm itself offers a promising path toward enabling more capable MPC controllers on resource-constrained robotic platforms.

More broadly, enabling optimization-based control on highly resource-constrained robotic systems requires addressing a multi-layer engineering challenge that spans several domains. First, the choice of optimization algorithm is critical, as different algorithms exhibit distinct convergence properties, numerical robustness, and computational structures. Selecting an algorithm that both converges reliably under reduced numerical precision and maps efficiently to specialized hardware architectures is therefore a key design decision.

Second, numerical representation plays a central role in embedded implementations. Hardware accelerators often rely on reduced-precision arithmetic, such as fixed-point representations, to achieve high performance and energy efficiency. However, reduced precision can affect the convergence behavior of optimization algorithms and requires careful analysis of numerical ranges, scaling strategies, and algorithm robustness.

Third, the computational architecture used to execute the solver must be carefully designed. Specialized computing substrates can exploit the structure of optimization algorithms to achieve high levels of parallelism and predictable execution timing. Mapping an optimization algorithm to such architectures requires the design of custom datapaths, memory structures, and communication interfaces that operate efficiently under strict resource constraints.

Finally, these algorithmic and architectural components must be integrated into a functioning robotic system. This includes the development of hardware interfaces, embedded software integra-

tion, and, in many cases, the design of custom electronics that physically integrate the accelerator with the robotic platform while respecting constraints on size, weight, and power consumption.

Taken together, these challenges make the deployment of optimization-based control on nano-scale robots a complex system-level problem that extends far beyond the design of efficient optimization algorithms alone. While many of the individual components required to build such systems exist in the literature, integrating them into a complete solution capable of operating reliably on a highly constrained robotic platform remains largely unexplored.

The central question addressed in this thesis is therefore whether optimization-based control can be executed fully onboard an extremely resource-constrained aerial robot while maintaining the high control frequencies required for stable flight.

1.2 Hardware-Aware MPC for Embedded Systems

Hardware–algorithm co-design provides a natural framework for addressing these challenges. In this approach, the structure of the optimization problem and the characteristics of the target computing platform are jointly considered during solver design. Rather than treating the optimization algorithm and the computing architecture as independent components, both are developed together to maximize computational efficiency and ensure reliable execution under strict resource constraints.

In the context of linear MPC, the quadratic programs that arise from the control formulation exhibit structured computational patterns that can be exploited to improve efficiency [18, 19]. First-order optimization methods such as the Alternating Direction Method of Multipliers (ADMM) are particularly attractive in this setting [19–21]. These algorithms rely on iterative update steps composed primarily of matrix–vector operations and projections onto simple constraint sets, resulting in a computational structure that is highly regular and predictable. Such characteristics make them well suited for implementation on specialized hardware architectures.

Field-Programmable Gate Arrays (FPGAs) provide a flexible platform for implementing such accelerators. By enabling custom datapaths that exploit fine-grained parallelism and deterministic execution timing, FPGAs allow the most computationally intensive components of the MPC optimization loop to be executed significantly faster than on a microcontroller alone [22, 23]. This capability makes FPGA-based acceleration particularly attractive for embedded robotic systems where strict timing guarantees and low-latency computation are essential.

Prior work on FPGA-based MPC has largely focused on applications such as industrial control systems, power electronics, or experimental setups based on large development platforms [24–37]. Many of these implementations target explicit MPC or rely on hardware platforms whose size, power consumption, and integration constraints differ substantially from those encountered in small robotic systems. As a result, the problem of deploying FPGA-accelerated MPC directly onboard a

mobile robot has remained largely unexplored.

To the best of the author's knowledge, this thesis presents the first implementation of linear MPC accelerated by FPGA hardware that is deployed and experimentally evaluated fully onboard a robotic platform. In particular, the proposed system demonstrates that constrained MPC can be executed fully onboard a nano-scale aerial robot. Achieving this required not only designing a hardware-accelerated optimization solver, but also addressing the broader system integration problem required to physically and computationally integrate the accelerator within the control pipeline of the robot.

Beyond FPGA-based acceleration, the limitations of current embedded computing platforms motivate the exploration of alternative computational substrates for optimization-based control. As robots become smaller and more energy-constrained, the ability to execute optimization algorithms with extremely low energy consumption becomes increasingly important. Neuromorphic computing architectures provide a fundamentally different computational paradigm that may offer new opportunities for implementing optimization algorithms with very high energy efficiency [38, 39].

While such platforms are not yet mature enough to be readily deployed within embedded robotic systems, they offer promising directions for future hardware substrates capable of executing optimization algorithms at extremely low power. For this reason, this thesis also investigates the feasibility of mapping optimization algorithms to neuromorphic hardware using Intel's Loihi 2 architecture. This study explores how the computational structure of optimization algorithms can be expressed within neuromorphic architectures, providing insight into how alternative computing paradigms might contribute to the long-term goal of enabling optimization-based control on extremely resource-constrained robotic platforms.

1.3 Contributions and Thesis Organization

The main contributions of this thesis are summarized as follows:

- The first experimental deployment of an FPGA-accelerated linear MPC controller fully onboard a robotic platform, demonstrating that optimization-based control can be executed in real time on an extremely resource-constrained nano-scale aerial robot.
- The design of an ADMM-based solver architecture tailored for reduced-precision hardware acceleration in embedded MPC applications.
- The design and fabrication of a custom FPGA expansion board enabling the physical integration of a hardware optimization accelerator with the Crazyflie nano-quadrotor platform.
- An experimental evaluation demonstrating substantial improvements in solver latency and energy–delay product compared to the TinyMPC baseline, enabling longer prediction hori-

zons, a greater number of solver iterations per control step, and reliable constraint handling within high-frequency control loops.

- An exploratory investigation of neuromorphic computing as a potential future hardware substrate for implementing optimization algorithms for control.

The remainder of this thesis is organized as follows.

- Chapter 2 reviews the principles of Model Predictive Control, the structure of the quadratic programs that arise in MPC, and the computational challenges associated with real-time optimization on embedded systems.
- Chapter 3 introduces the Crazyflie quadrotor platform, derives the linear system model used for control, and formulates the MPC optimization problem.
- Chapter 4 presents the ADMM-based solver architecture and describes its implementation using FPGA acceleration and hardware–software co-design.
- Chapter 5 presents the experimental evaluation of the proposed controller, including benchmarking of solver performance and real-world experiments on the Crazyflie platform performing constrained control tasks.
- Chapter 6 explores the feasibility of mapping optimization algorithms to neuromorphic hardware using Intel's Loihi 2 architecture.
- Finally, Chapter 7 summarizes the contributions of the thesis and outlines directions for future research.

1.3.1 Code and Hardware Repositories

All software and hardware artifacts developed as part of this thesis are released as open-source repositories to support reproducibility and further research. The main components of the system are available at the following locations:

- **PCB design**: Custom FPGA expansion board for the Crazyflie platform.
  `https://github.com/A2R-Lab/Crazyflie_FPGA_Deck`
- **FPGA solver implementation**: Includes the Python reference model, code generation tools, the Vitis HLS implementation of the ADMM solver, and the Vivado integration used to generate the final FPGA bitstream.
  `https://github.com/A2R-Lab/ADMM_FPGA`

- **Crazyflie firmware**: Modified Crazyflie firmware enabling communication with the FPGA accelerator and integration of the MPC controller within the flight stack.
  `https://github.com/A2R-Lab/crazyflie_fpga_firmware`
- **ROS2 interface**: ROS2 integration for motion capture, trajectory streaming, and high-level control of the Crazyflie platform.
  `https://github.com/A2R-Lab/ros2-crazyflie-mocap`

Chapter 2

Background

2.1 Model Predictive Control

Model Predictive Control (MPC) is a control strategy that computes control inputs by solving a constrained optimal control problem over a finite prediction horizon. At each sampling instant, the controller predicts the future evolution of the system using a dynamical model and selects a sequence of control inputs that minimizes a cost function while satisfying constraints on states and inputs.

Only the first control input of the optimal sequence is applied to the system. At the next sampling instant, the optimization problem is solved again using updated state measurements. This strategy, commonly referred to as *receding horizon control*, allows the controller to continuously adapt to disturbances and modeling errors while explicitly accounting for system constraints [1–3].

MPC is particularly attractive for robotic systems and safety-critical applications because constraints can be incorporated directly into the control design. These constraints may represent actuator limits, safety boundaries, or physical restrictions of the system.

2.1.1 System Modeling for MPC

The prediction model used by MPC is typically expressed in discrete-time state-space form

$$x_{k+1} = f_d(x_k, u_k) \tag{2.1}$$

where $x_k \in \mathbb{R}^n$ denotes the system state at time step k and $u_k \in \mathbb{R}^m$ denotes the control input.

For many control applications, the dynamics can be approximated using a linear time-invariant model

$$x_{k+1} = Ax_k + Bu_k \tag{2.2}$$

where $A \in \mathbb{R}^{n\times n}$ and $B \in \mathbb{R}^{n\times m}$ are the system matrices.

Although many physical systems exhibit nonlinear dynamics, linear models are often obtained by linearizing the system around a nominal operating point. The resulting linear approximation can provide sufficient accuracy for control design within a limited operating region.

2.1.2 Finite-Horizon Optimal Control

In MPC, the control problem is formulated over a finite prediction horizon of length N. At each sampling instant, the controller computes a sequence of control inputs that minimizes a cost function while respecting system dynamics and constraints.

A general nonlinear MPC problem can be written as

$$\begin{aligned} \min_{u_0,\dots,u_{N-1}} \quad & \sum_{k=0}^{N-1} \ell(x_k, u_k) + \ell_f(x_N) \\ \text{s.t.} \quad & x_{k+1} = f_d(x_k, u_k), \quad x_0 = \hat{x}_k \\ & x_k \in \mathscr{X}, \quad u_k \in \mathscr{U}. \end{aligned} \tag{2.3}$$

Here $\ell(\cdot)$ and $\ell_f(\cdot)$ denote the stage and terminal cost functions, $\hat{x}_k$ is the estimated system state at the current sampling instant, and $\mathscr{X}$ and $\mathscr{U}$ represent admissible sets for states and inputs.

When the system dynamics are linear, the model takes the standard discrete-time state-space form given in (2.2).

Even when the true system is nonlinear, it is common to approximate the dynamics using a linear model obtained by linearizing the system around a nominal operating point. The resulting linear time-invariant approximation can provide sufficient accuracy for control within a limited operating region.

If the dynamics are linear, the cost function is quadratic, and $\mathscr{X}, \mathscr{U}$ are convex polytopes, the MPC problem becomes a convex quadratic program of the form

$$\begin{aligned} \min_{u_0,\dots,u_{N-1}} \quad & \sum_{k=0}^{N-1} (x_k^\top Q x_k + u_k^\top R u_k) + x_N^\top P x_N \\ \text{s.t.} \quad & x_{k+1} = A x_k + B u_k, \\ & x_k \in \mathcal{X}, \quad u_k \in \mathcal{U}. \end{aligned} \tag{2.4}$$

Where Q, R, and P denote the state, input, and terminal-state weighting matrices, respectively.

This formulation results in a convex optimization problem that can be solved efficiently using quadratic programming algorithms [5, 6].

2.2 Quadratic Programs in MPC

Many MPC formulations ultimately require solving a problem that can be written as a quadratic program (QP) of the form

$$\begin{aligned} \min_{z} \quad & \tfrac{1}{2} z^\top H z + q^\top z \\ \text{s.t.} \quad & Gz \le h \\ & Az = b. \end{aligned} \tag{2.5}$$

Here z denotes the stacked vector of decision variables. The terms H and q define the quadratic and linear components of the objective function, respectively. The matrices G and A together with the vectors h and b encode the inequality and equality constraints of the optimization problem. If the matrix H is positive semidefinite, the problem is convex and any local optimum is also globally optimal.

2.2.1 Optimality Conditions

The Lagrangian associated with the quadratic program is

$$\mathcal{L}(z, \lambda, \mu) = \frac{1}{2} z^\top H z + q^\top z + \lambda^\top (Az - b) + \mu^\top (Gz - h) \tag{2.6}$$

where λ and μ denote the Lagrange multipliers corresponding to equality and inequality constraints.

The optimal solution satisfies the Karush–Kuhn–Tucker (KKT) conditions

$$
\begin{aligned}
\textbf{Stationarity:} \quad & Hz + q + A^\top \lambda + G^\top \mu = 0 \\
\textbf{Primal feasibility:} \quad & Az = b, \qquad Gz \leq h \\
\textbf{Dual feasibility:} \quad & \mu \geq 0 \\
\textbf{Complementarity:} \quad & \mu_i (G_i z - h_i) = 0 \ \ \forall i.
\end{aligned}
\tag{2.7}
$$

These conditions characterize optimal solutions and form the basis of many numerical algorithms used to solve quadratic programs [6].

2.3 Optimization Algorithms for Embedded MPC

Several algorithmic approaches exist for solving quadratic programs arising in MPC.

2.3.1 Active-Set Methods

Active-set methods iteratively estimate which inequality constraints are active at the solution and solve a sequence of equality-constrained subproblems [40]. These methods can be efficiently warm-started, which makes them attractive for MPC applications where successive optimization problems are closely related.

However, the computational complexity can vary significantly depending on the number of active constraints and problem size.

2.3.2 Interior-Point Methods

Interior-point methods handle inequality constraints using barrier functions and solve a sequence of Newton systems that converge to the optimal solution. These methods provide strong theoretical guarantees and are widely used in general-purpose optimization solvers [41].

However, their application in embedded settings is often limited by the computational and memory requirements associated with solving large Karush–Kuhn–Tucker (KKT) systems at each iteration. In addition, interior-point methods are generally less amenable to efficient warm-starting and fixed-iteration execution, which makes them less suitable for real-time control applications on resource-constrained platforms.

2.3.3 First-Order Methods

First-order methods rely primarily on gradient information and inexpensive iterations. Among these methods, operator splitting techniques such as the Alternating Direction Method of Multipliers (ADMM) have attracted significant interest for embedded optimization [20, 42].

ADMM decomposes the optimization problem into simpler subproblems that can often be solved efficiently. The resulting algorithm typically involves matrix-vector multiplications, projections onto constraint sets, and vector updates. These operations are well suited for parallel execution and hardware acceleration.

Because first-order methods trade solution accuracy for computational simplicity, they are particularly attractive in real-time control applications where moderate solution accuracy is sufficient.

2.4 Real-Time Optimization Challenges

Unlike controllers such as PID or LQR, which compute control inputs through closed-form feedback laws, MPC requires solving an optimization problem at each control step. For systems with fast dynamics, the available time to compute the control input can be extremely limited.

The computational cost of MPC depends on several factors, including the prediction horizon, the dimension of the system state, and the number of constraints. As these parameters increase, the complexity of the optimization problem grows, making real-time execution more challenging.

Historically, MPC was first deployed in industrial process control applications where system dynamics were relatively slow and control cycles occurred on the order of seconds or minutes [4]. In modern robotic systems, control frequencies can reach hundreds or thousands of Hertz, placing significantly stricter timing constraints on the optimization process [1].

At the same time, improvements in single-thread CPU performance have slowed despite continued increases in transistor density, largely due to power and frequency scaling limits in modern processors [43]. As a result, algorithmic efficiency and hardware-aware implementations are increasingly important for achieving real-time MPC performance.

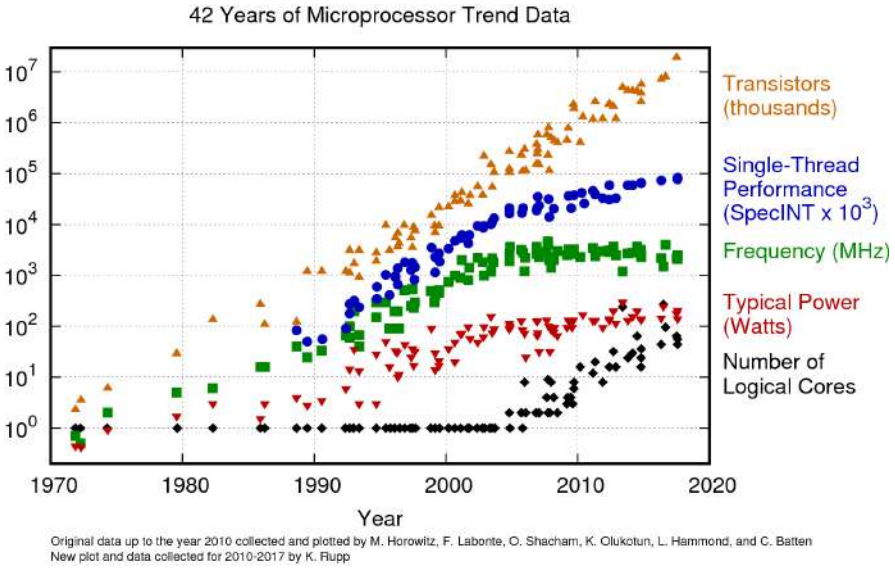


Figure 2.1: Growth in transistor density has continued according to Moore's law, while improvements in single-thread CPU performance have slowed.

2.5 Hardware Platforms for Real-Time Optimization

Different computing architectures offer different trade-offs for implementing optimization algorithms.

2.5.1 Microcontrollers and CPUs

Central Processing Units (CPUs) are general-purpose processors capable of executing complex control algorithms with strong software support [44]. In embedded systems, microcontrollers (MCUs) represent a class of low-power CPUs widely used in robotics and control applications [45, 46].

Nevertheless, specialized algorithmic approaches have enabled MPC deployment even on very resource-constrained platforms. At the extreme end of embedded systems, *TinyMPC* demonstrates linear MPC on microcontrollers by precomputing and caching matrix factorizations and using division-free updates, enabling real-time QP solving on devices with very limited computational resources [17, 47]. While this shows that MPC can run onboard small quadrotors, the microcontroller is often heavily utilized, leaving limited computational headroom for sensing, estimation, or higher-level tasks.

2.5.2 Graphics Processing Units

Graphics Processing Units (GPUs) provide massive data-level parallelism by executing the same instruction across large arrays of processing units [48–50]. This architecture makes GPUs particularly well suited for linear algebra operations and large-scale optimization problems.

Several works have demonstrated GPU-accelerated MPC implementations. For example, ADMM-based solvers have been mapped efficiently to GPU architectures [51], and real-time nonlinear MPC implementations exploiting GPU parallelism have been reported for high-performance robotic applications [52, 53]. While GPUs can provide substantial computational throughput, their power consumption and hardware requirements often make them less suitable for small embedded platforms.

2.5.3 Field-Programmable Gate Arrays

Field-Programmable Gate Arrays (FPGAs) allow the implementation of custom hardware pipelines tailored to specific algorithms [22, 23, 54, 55]. By exploiting fine-grained parallelism and deterministic execution, FPGAs can achieve very low latency for linear algebra and optimization routines.

Between the resource-constrained microcontroller implementations and high-throughput GPU solutions, FPGAs offer a middle ground for embedded MPC deployment. Fixed-point MPC solvers can be implemented directly in hardware and executed in parallel with a host processor, offloading the most computationally intensive operations while remaining compatible with small robotic platforms. In addition, learning-based controllers, including neural network policies trained through reinforcement learning and deployed on FPGAs, have been explored for agile quadrotor control, trading some of the guarantees of model-based MPC for improved adaptability in uncertain environments [56].

2.5.4 Emerging Computing Platforms

Emerging computing platforms, including neuromorphic processors and photonic computing systems, have also been investigated for accelerating optimization algorithms [57]. These architectures offer alternative computational paradigms that may enable new approaches to solving optimization problems.

Neuromorphic processors, for example, implement networks of spiking neurons that communicate through asynchronous events and operate in a highly parallel and event-driven fashion [38, 39]. Optimization algorithms can in some cases be mapped to neural dynamics, allowing the state evolution of the network to converge toward an optimal solution.

Although promising, these technologies remain relatively immature and often require specialized programming models and hardware–software co-design. In particular, efficiently expressing conventional optimization algorithms on neuromorphic hardware typically requires significant reformulation of the algorithm to match the underlying neural computation model.

Chapter 3

System Modeling and MPC Problem Formulation

This chapter defines the dynamical model of the quadrotor and formulates the Model Predictive Control (MPC) problem used throughout the thesis. The quadcopter platform is first introduced as a representative example of a fast, constrained, and resource-limited control system. The nonlinear dynamics of the vehicle are then presented together with their linearization around a hovering operating point. Finally, the linear MPC problem is formulated as a quadratic program whose structure is later exploited in the hardware-oriented solver implementations.

3.1 Quadrotor Control as a Benchmark Problem

Quadrotor control has become a widely studied problem in control theory and robotics due to both its practical relevance and the challenging characteristics of the underlying dynamics [17, 58–65]. A quadrotor is a rotary-wing unmanned aerial vehicle equipped with four propellers arranged in a cross configuration. By varying the thrust produced by each rotor, the vehicle can generate forces and torques that control its motion in space.

The system is underactuated: the vehicle has six degrees of freedom (three translational and three rotational), but only four independent control inputs corresponding to the individual rotor thrusts. As a result, translational motion is indirectly achieved by changing the vehicle orientation, which tilts the thrust vector and produces horizontal acceleration. The dynamics are nonlinear and strongly coupled, and the open-loop system is unstable, making feedback control essential for stable flight.

In addition to these dynamic properties, quadrotors are subject to physical constraints such as actuator saturation, limits on angular rates, and safety constraints on the workspace. These charac-

teristics make quadrotors a representative testbed for constrained optimal control methods such as MPC.

Model Predictive Control is particularly well suited to this setting because it allows the explicit incorporation of constraints while predicting the future evolution of the system. However, achieving real-time MPC performance on small aerial robots is challenging due to the limited computational resources available on board and the high control frequencies required for stable flight.

In this thesis, the quadrotor serves as the primary application for evaluating hardware-oriented MPC implementations. The Crazyflie nano-quadrotor platform is used as the experimental system, and a linearized model of its dynamics is employed to formulate the MPC problem solved in later chapters.

3.2 Nonlinear Quadrotor Dynamics

The quadrotor dynamics are modeled using a rigid-body representation that captures both translational and rotational motion. The state vector consists of the vehicle position, orientation, linear velocity, and angular velocity. The control inputs correspond to the thrust generated by each rotor.

The system state is defined as

$$x = \begin{bmatrix} p \\ q \\ v \\ \omega \end{bmatrix} \in \mathbb{R}^{13}, \qquad u = \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \end{bmatrix} \in \mathbb{R}^4, \tag{3.1}$$

where $p \in \mathbb{R}^3$ represents the position in the world frame, $q \in \mathbb{R}^4$ is the unit quaternion describing orientation, $v \in \mathbb{R}^3$ is the linear velocity, and $\omega \in \mathbb{R}^3$ is the angular velocity expressed in the body frame.

The nonlinear dynamics follow the formulation presented in [66] and adopted in TinyMPC [17]. The model parameters correspond to those identified for the Crazyflie 2.1 Brushless platform in [67].

3.2.1 Translational Dynamics

The translational dynamics describe the evolution of the position and linear velocity of the quadrotor:

$$
\begin{aligned}
\dot{p} &= v, \\
\dot{v} &= g_{vec} + \frac{k_t}{m} Q(q) \begin{bmatrix} 0 \\ 0 \\ u_1 + u_2 + u_3 + u_4 \end{bmatrix},
\end{aligned}
\tag{3.2}
$$

where m denotes the vehicle mass, k_t is the thrust coefficient, and $g_{vec} = [0, 0, -g]^\top$ represents the gravitational acceleration. The matrix $Q(q)$ maps the thrust generated by the rotors to accelerations expressed in the world frame.

3.2.2 Rotational Dynamics

The rotational motion of the quadrotor is described by quaternion kinematics and rigid-body rotational dynamics:

$$
\begin{aligned}
\dot{q} &= \frac{1}{2} L(q) H \omega, \\
\dot{\omega} &= J^{-1} \left(-\hat{\omega} J \omega + \tau_{\text{motors}} \right),
\end{aligned}
\tag{3.3}
$$

where J is the inertia matrix and $\hat{\omega}$ denotes the skew-symmetric matrix associated with the angular velocity vector. The matrix

$$
L(q) = \begin{bmatrix} s & -v^\top \\ v & sI_3 + \hat{v} \end{bmatrix}, \qquad q = [s, v^\top]^\top
\tag{3.4}
$$

is the quaternion kinematic matrix, while

$$
H = \begin{bmatrix} 0_{1\times 3} \\ I_3 \end{bmatrix}
\tag{3.5}
$$

maps angular velocity to quaternion derivatives.

The torque generated by the motors is given by

$$
\tau_{\text{motors}} = \begin{bmatrix} -\ell k_t (u_1 + u_2 - u_3 - u_4) \\ -\ell k_t (u_1 - u_2 - u_3 + u_4) \\ -k_m u_1 + k_m u_2 - k_m u_3 + k_m u_4 \end{bmatrix},
\tag{3.6}
$$

where ℓ denotes the arm length of the quadrotor and k_m is the rotor torque coefficient.

3.3 Linearization Around Hover

To enable the use of linear MPC, the nonlinear dynamics are linearized around a nominal operating point corresponding to hovering flight. The reference state x_{ref} corresponds to a stationary vehicle with zero velocity and level orientation, while the reference input u_{ref} corresponds to the rotor thrust required to balance gravity.

The discrete-time dynamics used by the controller are obtained by integrating the continuous-time model using a fourth-order Runge–Kutta (RK4) scheme. The linearization is then computed by evaluating the Jacobians of the discrete-time dynamics at the reference operating point:

$$A_{full} = \frac{\partial f_{\text{RK4}}}{\partial x}\Big|_{x_{\text{ref}},u_{\text{ref}}}, \qquad B_{full} = \frac{\partial f_{\text{RK4}}}{\partial u}\Big|_{x_{\text{ref}},u_{\text{ref}}}. \tag{3.7}$$

Because the state representation includes quaternions, a coordinate transformation is applied to express orientation errors using a minimal three-dimensional representation. This transformation reduces the state dimension from 13 to 12 and yields the linearized system

$$x^{+} = Ax + Bu, \tag{3.8}$$

with

$$A = E(q_{\text{ref}})^{\top} A_{full} E(q_{\text{ref}}), \qquad B = E(q_{\text{ref}})^{\top} B_{full}. \tag{3.9}$$

The transformation matrix is defined as

$$E(q) = \text{diag}(I_3, L(q)H, I_6) \in \mathbb{R}^{13\times 12}. \tag{3.10}$$

The resulting linear system captures the local dynamics around the hover condition and forms the prediction model used in the MPC formulation.

3.4 Linear MPC Formulation

Given the linearized system dynamics, the control problem is formulated as a finite-horizon optimal control problem. Let H denote the prediction horizon and let $x_k \in \mathbb{R}^n$ and $u_k \in \mathbb{R}^m$ represent the state and control input at time step k.

The objective is to minimize deviations from the desired state x_{ref} and control input u_{ref} over the prediction horizon:

$$\begin{aligned}
\min_{u_0,\dots,u_{H-1}} \quad & \sum_{k=0}^{H-1} (x_k - x_{\text{ref}})^\top Q (x_k - x_{\text{ref}}) + (u_k - u_{\text{ref}})^\top R (u_k - u_{\text{ref}}) \\
& + (x_H - x_{\text{ref}})^\top Q_f (x_H - x_{\text{ref}}) \\
\text{s.t.} \quad & x_{k+1} = A x_k + B u_k, \\
& x_0 = x_{\text{init}}, \\
& x_k \in \mathscr{X}, \quad k = 0, \dots, H, \\
& u_k \in \mathscr{U}, \quad k = 0, \dots, H-1.
\end{aligned} \tag{3.11}$$

The matrices Q, R, and Q_f define the relative weighting of state deviations and control effort.

3.4.1 Quadratic Program Formulation

The optimization problem can be written as a quadratic program by stacking all states and inputs into a single decision vector. Using an interleaved ordering, the decision variable is defined as

$$z = \begin{bmatrix} x_0 \\ u_0 \\ x_1 \\ u_1 \\ \vdots \\ x_H \end{bmatrix}. \tag{3.12}$$

The quadratic cost function yields a block-diagonal Hessian matrix

$$P = \begin{bmatrix} Q & & & \\ & R & & \\ & & \ddots & \\ & & & Q_f \end{bmatrix}. \tag{3.13}$$

The system dynamics are enforced through equality constraints of the form

$$x_{k+1} - A x_k - B u_k = 0, \tag{3.14}$$

which can be assembled into a linear constraint matrix

$$A_{\text{eq}} z = b_{\text{eq}}. \tag{3.15}$$

3.4.2 Inequality Constraints

Input and state constraints can be represented in the general linear form

$$\begin{aligned} l_u \le A_u u_k \le u_u, \qquad & k = 0, \dots, H-1, \\ l_x \le A_x x_k \le u_x, \qquad & k = 0, \dots, H. \end{aligned} \tag{3.16}$$

This formulation can represent a variety of constraints, including actuator limitations such as bounds on the rotor thrusts, as well as state constraints arising from safety or task requirements.

The resulting constraints, including both the system dynamics and the inequality constraints, can be stacked together and written compactly as

$$l \le Az \le u, \tag{3.17}$$

where the matrix A contains all equality and inequality constraint matrices. Equality constraints are represented by setting the corresponding entries of the lower and upper bounds equal, i.e., $l_i = u_i$.

3.5 Structure of the Resulting Optimization Problem

The MPC formulation described above results in a sparse quadratic program whose structure depends on the system dimension and the prediction horizon. The Hessian matrix is block diagonal, while the dynamics constraints produce a banded sparsity pattern in the constraint matrix.

This structure plays a crucial role in the design of efficient solvers. In particular, the repeated structure of the dynamics and the fixed sparsity pattern of the KKT system enable precomputation and hardware acceleration techniques. These properties are exploited in later chapters to design FPGA-based implementations of the ADMM algorithm for real-time MPC execution.

3.6 The ADMM Algorithm

The quadratic program obtained from the MPC formulation must be solved repeatedly at every control step. To enable real-time execution on resource-constrained hardware, an algorithm is required that exploits the structure of the problem and allows efficient implementation.

In this work, the optimization problem is solved using the *alternating direction method of multipliers* (ADMM) [20]. ADMM is an operator-splitting method that decomposes constrained optimization problems into a sequence of simpler subproblems, making it particularly suitable for large-scale or structured problems such as those arising in MPC. The algorithm is also widely used in modern optimization solvers, including OSQP [19].

3.6.1 ADMM Formulation

ADMM solves optimization problems of the form

$$\begin{aligned} \min_{x,z} \quad & f(x) + g(z) \\ \text{s.t.} \quad & Ax + Bz = c, \end{aligned} \tag{3.18}$$

where $x \in \mathbb{R}^n$ and $z \in \mathbb{R}^m$ are decision variables, and $A \in \mathbb{R}^{p \times n}$, $B \in \mathbb{R}^{p \times m}$, $c \in \mathbb{R}^p$. The functions f and g are assumed to be closed, proper, and convex.

Many constrained optimization problems, including the quadratic programs arising in MPC, can be expressed in this form by introducing auxiliary variables that separate the objective function from the constraints.

3.6.2 Augmented Lagrangian

To apply ADMM, the augmented Lagrangian associated with the equality constraint is introduced:

$$L_\rho(x, z, y) = f(x) + g(z) + y^\top (Ax + Bz - c) + \frac{\rho}{2} \| Ax + Bz - c \|_2^2, \tag{3.19}$$

where $y \in \mathbb{R}^p$ is the dual variable associated with the constraint and $\rho > 0$ is a penalty parameter.

The quadratic penalty term improves numerical stability and enforces the satisfaction of the equality constraint during the iterative optimization process.

3.6.3 ADMM Iterations

Starting from initial values (x^k, z^k, y^k), the ADMM iterations consist of the following updates:

$$\begin{aligned} x^{k+1} &:= \arg\min_x L_\rho(x, z^k, y^k), \\ z^{k+1} &:= \arg\min_z L_\rho(x^{k+1}, z, y^k), \\ y^{k+1} &:= y^k + \rho(Ax^{k+1} + Bz^{k+1} - c). \end{aligned} \tag{3.20}$$

Each iteration therefore alternates between minimizing the augmented Lagrangian with respect to the primal variables and updating the dual variable.

The quantity

$$r^{k+1} = Ax^{k+1} + Bz^{k+1} - c$$

is referred to as the *primal residual* and measures the violation of the equality constraint.

3.6.4 Convergence

Under standard convexity assumptions, ADMM converges to a primal–dual optimal solution [20]:

$$x^k \to x^\star, \qquad z^k \to z^\star, \qquad y^k \to y^\star.$$

Although ADMM is a first-order method, it is often highly effective when the resulting subproblems admit efficient closed-form solutions or simple linear solves. These characteristics make ADMM particularly attractive for embedded implementations of optimization-based control.

3.6.5 Application to Quadratic Programs

Consider the quadratic program introduced in 2.5:

$$\begin{aligned} \min_x \quad & \tfrac{1}{2} x^\top H x + q^\top x \\ \text{s.t.} \quad & \ell \le Ax \le u. \end{aligned} \tag{3.21}$$

This problem can be rewritten by introducing an auxiliary variable z:

$$\begin{aligned} \min_{x,z} \quad & \tfrac{1}{2} x^\top H x + q^\top x + I_{[\ell,u]}(z) \\ \text{s.t.} \quad & Ax = z, \end{aligned} \tag{3.22}$$

where $I_{[\ell,u]}$ is the indicator function of the interval $[\ell, u]$,

$$I_{[\ell,u]}(z) = \begin{cases} 0 & \text{if } \ell \le z \le u, \\ \infty & \text{otherwise.} \end{cases} \tag{3.23}$$

The augmented Lagrangian becomes

$$L_\rho(x, z, y) = \tfrac{1}{2} x^\top H x + q^\top x + I_{[\ell,u]}(z) + y^\top (Ax - z) + \tfrac{\rho}{2} \|Ax - z\|_2^2. \tag{3.24}$$

x-update

Minimizing the augmented Lagrangian with respect to x yields

$$x^{k+1} = \arg\min_x L_\rho(x, z^k, y^k). \tag{3.25}$$

Setting the gradient to zero gives

$$Hx + q + A^\top y + \rho A^\top (Ax - z) = 0, \tag{3.26}$$

which leads to the linear system

$$(H + \rho A^\top A) x^{k+1} = -q - A^\top y^k + \rho A^\top z^k. \tag{3.27}$$

The coefficient matrix $(H + \rho A^\top A)$ remains constant across ADMM iterations and, in the case of linear MPC, across successive control problems. As a result, its factorization can be precomputed, allowing the linear system to be solved efficiently at each iteration. This strategy is exploited in existing solvers such as OSQP [19].

z-update

The z-update minimizes the augmented Lagrangian with respect to z:

$$z^{k+1} = \arg\min_z \; I_{[\ell,u]}(z) - y^\top z + \frac{\rho}{2} \|Ax^{k+1} - z\|_2^2. \tag{3.28}$$

This minimization reduces to a projection onto the box $[\ell, u]$:

$$z^{k+1} = \Pi_{[\ell,u]} \left(Ax^{k+1} + \frac{1}{\rho} y^k \right), \tag{3.29}$$

where $\Pi_{[\ell,u]}(\cdot)$ denotes elementwise projection onto the interval.

y-update

Finally, the dual variable is updated according to

$$y^{k+1} = y^k + \rho(Ax^{k+1} - z^{k+1}). \tag{3.30}$$

Together, these steps define the standard ADMM iteration for solving quadratic programs with box constraints.

3.7 Closed-Loop MPC Execution with ADMM

The quadratic program derived in the previous sections must be solved repeatedly during operation of the controller. At each control step, the current state of the quadrotor is estimated, the current reference setpoint is provided to the controller, and a new optimal control problem is solved to determine the control input applied to the motors.

In a typical MPC implementation, the optimization problem is solved at every control step using the current state estimate as the initial condition of the prediction model and the current reference setpoint as the target to be tracked over the horizon. The resulting control sequence is then applied in a receding-horizon fashion: only the first control input is executed, and the optimization is repeated at the next sampling instant.

Algorithm 1 summarizes the overall structure of the control loop when the quadratic program is solved using ADMM.

The algorithm highlights the two nested loops that characterize optimization-based control. The outer loop corresponds to the real-time control cycle, which interacts with the estimator, the reference generator, and the physical system. The inner loop corresponds to the iterative optimization routine used to solve the quadratic program.

The computational cost of this control scheme is dominated by the repeated ADMM iterations required to solve the optimization problem at each control step. Consequently, improving the efficiency of these iterations can significantly increase the achievable control frequency or reduce the computational resources required for onboard execution.

Figure 3.1 illustrates the high-level organization of the closed-loop controller. The estimator provides the current state estimate, the reference setpoint specifies the desired motion, the ADMM-based MPC controller computes the control sequence, and the first control input is sent to the motors.

Algorithm 1 Closed-loop linear MPC using ADMM

Initialize MPC problem data and ADMM variables
while controller is running **do**
    Acquire current state estimate x_{est} from the estimator
    Acquire current reference setpoint q
    Update the initial-condition constraint $x_0 = x_{\text{est}}$
    Update the reference-dependent terms using q
    Warm-start optimization variables
    while not converged and time budget not exceeded **do**
        x-update: solve for the predicted trajectory variables — eq. (3.27)
        z-update: project onto the constraint set — eq. (3.29)
        y-update: update the dual variables — eq. (3.30)
    end while
    Extract first control input $u_0^\star$
    Apply $u_0^\star$ to the motors
end while

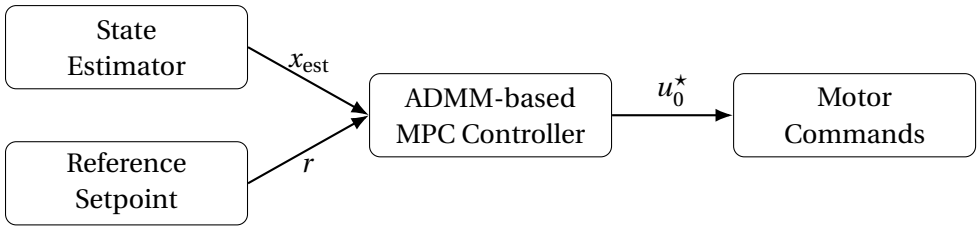


Figure 3.1: High-level structure of the closed-loop MPC controller.

The following chapters investigate how the ADMM iterations described in Section 3.6 can be implemented efficiently on specialized hardware platforms. In particular, the focus is on accelerating the iterative optimization loop to enable real-time MPC execution on resource-constrained systems.

Chapter 4

Hardware–Algorithm Co-Design and FPGA Implementation

In this chapter, we describe the complete hardware-–algorithm co-design process leading to the implementation of the ADMM algorithm on an FPGA and its deployment on a custom-designed PCB mounted on a Crazyflie platform, enabling efficient real-time flight control.

The chapter first motivates the use of FPGA acceleration for embedded MPC and positions the present work within the existing literature. It then describes the hardware platform selection process, the FPGA-oriented reformulation of the solver, and the structure-exploiting algorithmic design that enables efficient hardware execution. The design and implementation of the custom PCB are subsequently presented, followed by system integration with the Crazyflie firmware stack. Finally, the performance of the FPGA implementation is evaluated through benchmarks and real-world flight experiments.

4.1 Motivation and Related Work

Linear quadratic programming (QP)-based model predictive control (MPC) is a well-established control approach, and numerous solvers exist for different platforms, ranging from small-footprint embedded devices to large-scale GPU systems [17, 51]. While these developments demonstrate the flexibility and maturity of modern MPC solvers, deploying optimization-based control on very small edge platforms remains challenging. In particular, nano-quadrotors such as the Crazyflie impose strict constraints on computational resources, memory capacity, power consumption, and onboard weight, which significantly limit the complexity of the optimization problem that can be solved in real time.

Recent work has demonstrated that MPC can be executed directly on the Crazyflie microcontroller. In particular, *TinyMPC* [17] implements an ADMM-based QP solver capable of running onboard a Crazyflie nano-quadrotor. This result represents an important step toward fully embedded optimization-based control on ultra-lightweight aerial robots. However, executing the entire optimization routine on the microcontroller consumes a substantial portion of the available computational budget, leaving limited headroom for other critical tasks such as state estimation, communication, safety monitoring, and higher-level autonomy. As the prediction horizon or update rate increases, these constraints become increasingly restrictive and can limit the practical operating envelope of the controller.

FPGA-based MPC implementations have also been explored in the literature [24–37]. However, most of these solutions either:

- rely on *explicit MPC* [32–35], which pre-computes control laws offline and stores them in memory [68]. While efficient for certain applications, explicit MPC is inflexible and scales poorly with dynamic constraints.
- target high throughput for large problem scales [28, 30], by deploying on very large FPGA boards, that are not suitable for embedded and edge deployment.
- are often designed for power electronics or general-purpose development boards [26, 29, 31], or use offboard, floating-point implementations [36], rather than robotics platforms with strict weight, power, and interface constraints.

In this work, these limitations are addressed by introducing an **FPGA accelerator for MPC** that operates alongside the Crazyflie microcontroller. Instead of executing the entire optimization routine on the MCU, the computationally intensive ADMM solver is offloaded to the FPGA. This hardware–software co-design allows the optimization to be performed with significantly reduced latency while preserving valuable microcontroller resources for the rest of the flight-control stack. At the same time, the additional computational capability provided by the FPGA enables the solution of larger optimization problems and the handling of constraints at update rates that would not be feasible using the MCU-only TinyMPC implementation.

Table 4.1 summarizes representative prior work and highlights the positioning of the present thesis. Prior FPGA-based MPC studies have mainly focused on generic benchmarks, simulation, or development-board implementations, rather than fully onboard deployment on a resource-constrained aerial robot. Conversely, prior onboard Crazyflie demonstrations have either relied on MCU-based MPC or on non-MPC controllers. To the best of the author's knowledge, this thesis is the first work to combine FPGA acceleration, online MPC, and fully onboard deployment on a Crazyflie-class nano-quadrotor.

Work	Solver	Application	Validation level	Onboard robot
Jerez et al. [28]	Sparse QP	Generic MPC	FPGA dev board evaluation	×
Hartley et al. [25]	PDIP	Aircraft control	FPGA-in-the-loop simulation	×
Jerez et al. [26]	FGM / ADMM	Generic MPC	FPGA dev board experiments	×
Shukla et al. [31]	Splitting-based MPC	Generic MPC	Code-generation framework	×
Jeong et al. [29]	ADMM	Power electronics	Hardware implementation	×
Azem et al. [56]	Neural controller	Crazyflie nano-quadrotor	FPGA flight experiments	✓
TinyMPC [17]	ADMM	Crazyflie nano-quadrotor	MCU flight experiments	✓
This thesis	**ADMM**	**Crazyflie nano-quadrotor**	**Custom FPGA PCB + flight tests**	✓

Table 4.1: Comparison of representative MPC implementations on FPGA and related embedded control systems.

The main contributions of this work can therefore be summarized as follows:

- Hardware–software co-design of an MPC solver and FPGA accelerator enabling efficient parallel computation of the ADMM iterations.
- Implementation of the solver using compressed matrix storage and fixed-point arithmetic tailored to FPGA execution.
- Design and fabrication of a custom PCB hosting the FPGA and interfacing directly with the Crazyflie platform.
- Deployment and experimental validation of the complete onboard system through real-world flight experiments.

4.2 Preliminary work

4.2.1 FPGA Selection and Sizing Considerations

The first step in the hardware–software co-design process is the selection of the target hardware for deployment. This choice is driven by a set of constraints and trade-offs, including available computational resources, communication latency, power consumption, and physical integration with the aerial platform.

FPGA resources and their relevance to MPC

FPGA capacity is usually described in terms of a few key resources.

- **Look-up tables (LUTs)** are the basic building blocks of programmable logic; they implement combinational functions and, in many FPGAs, can also act as small memories. The number of LUTs limits how much control logic, address generation, and state machines can be implemented.
- **DSP blocks** (digital signal processing slices) are hard macros optimized for multiply accumulate operations (e.g., $a \cdot b + c$ in one or a few cycles). For MPC, matrix-vector products and linear algebra dominate the cost, so DSP count directly bounds achievable throughput and parallelism.
- **Block RAM (BRAM)** provides on-chip storage for coefficients, state, and intermediate vectors; insufficient BRAM forces data to move off-chip, increasing latency and limiting speed. For an iterative solver like ADMM, all three resources matter: LUTs for the overall datapath and control, DSPs for the linear algebra, and BRAM for storing the system matrices and iteration state.

Table 4.2 summarizes representative FPGAs grouped by the three deployment categories considered in this work. Orders-of-magnitude differences in resources explain why very small FPGAs can only support heavily reduced problem sizes or simplified solvers, while larger devices allow longer horizons and higher precision at the cost of board size and power.

Category	**Device**	**LUTs**	**DSPs**	**Block RAM**
Offboard, bulky	Xilinx Kintex-7 325T [55]	~326 k	900	~16 Mb
	Xilinx Zynq UltraScale+ [69]	~274 k	1,968	~27 Mb
Onboard, medium	Xilinx Artix-7 100T [55] (selected)	~101 k	240	~8.6 Mb
Onboard, small	Lattice iCE40UP5K [70] (Crazyflie deck)	~5.3 k	8	120 Kb

Table 4.2: Comparison of representative FPGAs by deployment category. Resource counts (LUTs, DSP slices, block RAM) determine the feasible problem size and solver complexity for MPC.

Three possible FPGA configurations were considered:

1. **Offboard, bulky FPGA:** A large FPGA is used offboard, relaxing resource constraints and allowing the implementation of larger optimization problems (e.g., longer horizons and larger matrices). Communication with the drone occurs wirelessly via radio, supported by the existing firmware.
2. **Onboard, small FPGA:** The implementation is constrained to fit on the very small FPGA available on the Crazyflie expansion deck. While attractive from a miniaturization perspective, this option is extremely challenging due to the severely limited computational and memory resources.

3. **Onboard, medium FPGA:** An FPGA larger than the ICE40UP5K is selected, while still remaining small enough to fit on a $\approx 3.5 \times 3.5$ cm PCB that can be mounted directly on the drone.

When using an offboard FPGA, communication latency introduced by the wireless link must be carefully accounted for. However, this option benefits from significantly increased computational resources, enabling the solution of larger optimization problems. Conversely, an onboard FPGA eliminates wireless communication latency, but constrains the problem size due to limited available resources.

The communication interface between the FPGA and the drone depends on the selected configuration:

- **Onboard FPGA:** the Crazyflie expansion interface exposes UART, I^2C, and SPI communication buses.
- **Offboard FPGA:** communication is performed via a USB radio dongle (Crazyradio 2.0, connected to a host computer that interfaces with the FPGA.

Communication interfaces and bandwidth

Onboard options differ in speed and protocol overhead.

- **UART** (Universal Asynchronous Receiver-Transmitter) is simple and widely used for debugging or low-rate telemetry; typical baud rates range from 115.2 kbps to about 3 Mbps on capable hardware [71].
- **I^2C** (Inter-Integrated Circuit) uses two wires and supports multiple devices on one bus; standard and fast modes run at 100 kbps and 400 kbps, with high-speed modes up to a few Mbps [72].
- **SPI** (Serial Peripheral Interface) is full-duplex, uses a dedicated clock, and has no fixed upper limit on clock rate—often 10–50 MHz on embedded links—so it offers the highest throughput and lowest latency for time-critical state and control exchange [73].
- **Radio** links such as the Crazyradio 2.0 (based on the nRF52840 chip [74]) use proprietary or BLE-style protocols with typical useful data rates in the 250 kbps–2 Mbps range, and add latency and jitter compared to wired buses.

Table 4.3 summarizes these trade-offs; the need for high-rate, low-latency exchange of state and control commands motivates the choice of SPI for the onboard FPGA–microcontroller link (Section 4.5.3).

UART	
Typical speed	115 kbps–3 Mbps
Notes	Asynchronous; simple; common for debug/telemetry
I^2C	
Typical speed	100–400 kbps (up to ~3.4 Mbps fast)
Notes	Two-wire; multi-device; moderate throughput
SPI	
Typical speed	10–50 Mbps (clock-dependent)
Notes	Full-duplex; low overhead; used for control loop
Crazyradio 2.0 (offboard)	
Typical speed	250 kbps–2 Mbps
Notes	Wireless; latency and jitter; link-dependent

Table 4.3: Typical bandwidth and characteristics of communication interfaces relevant to FPGA–drone connectivity. SPI offers the highest wired throughput and is used for the onboard control loop.

After evaluating all configurations, the onboard medium FPGA solution was selected for the following reasons:

- While an offboard FPGA would simplify the implementation, it would conflict with the primary objective of this work: demonstrating that linear MPC can be solved on low-power edge hardware.
- The very small FPGA available on the existing expansion deck (ICE40UP5K) presents an appealing challenge; however, the level of optimization required to implement ADMM on such constrained hardware would significantly exceed the scope of a master's thesis. Furthermore, extending the controller with additional functionalities, such as obstacle avoidance, would be extremely difficult on this platform. Finally, the lack of mature high-level synthesis (HLS) tools for this FPGA further limits its practicality.

Based on these considerations, an **AMD Xilinx Artix-7 100T** FPGA was selected. This device is widely available, supported by mature development tools with HLS capabilities, and provides sufficient resources to accommodate the ADMM-based controller while enabling future extensions beyond basic tracking control.

4.2.2 High-Level Synthesis

High-Level Synthesis (HLS) is a design methodology that enables the generation of hardware descriptions from high-level programming languages such as C, C++, or SystemC. Instead of manually

describing cycle-accurate behavior using Register-Transfer Level (RTL) languages (e.g., Verilog or VHDL), the designer specifies the algorithm at a higher level of abstraction, while the HLS tool automatically derives the corresponding hardware implementation, including datapaths, control logic, and memory interfaces.

Compared to traditional RTL-based design, HLS significantly reduces development complexity and design time. This abstraction allows the designer to focus primarily on algorithmic correctness and architectural exploration, rather than low-level hardware details. In particular, HLS facilitates rapid iteration, enabling quick evaluation of different design choices such as loop unrolling, pipelining, and memory partitioning through simple code annotations and pragmas.

In this work, HLS was chosen for several key reasons:

- First, it substantially lowers the barrier to entry for FPGA-based acceleration, making the design process more accessible and manageable for a single developer. Implementing a sophisticated optimization algorithm such as ADMM directly in RTL would require a significant engineering effort, extensive debugging, and deep expertise in low-level hardware design. Achieving a functionally correct and high-performance RTL implementation of such an algorithm would typically demand a large, specialized team and a long development cycle.
- Second, HLS provides greater flexibility during the hardware–software co-design process. Since the hardware description is derived from a high-level algorithmic specification, modifications to the problem formulation, numerical precision, or control structure can be implemented rapidly without rewriting large portions of RTL code. This flexibility is particularly valuable in research-oriented projects, where algorithmic changes are frequent and design requirements may evolve over time.
- Finally, the use of HLS aligns with one of the broader objectives of this work: to demonstrate that efficient and high-performance hardware–software co-design does not necessarily require a large engineering team or industrial-scale resources. By relying on HLS, it is possible for a single researcher to develop, optimize, and deploy a complex FPGA-based control algorithm that achieves real-time performance on resource-constrained edge hardware. In contrast, an equivalent RTL-based approach would be significantly more difficult to realize within the scope of an individual master's thesis, especially for an algorithm of comparable sophistication.

4.3 Algorithm Co-Design

The algorithm implemented in this work is ADMM, as described in Section 3.6.5. For clarity, the iteration steps can be summarized as follows:

$$
\begin{aligned}
&\text{Solve linear system: } K \cdot x^{+} = A^{\top}(\rho z - y) \\
&z^{+} = \Pi_{[\ell,u]}\left(A \cdot x^{+} + \frac{y}{\rho}\right) \\
&y^{+} = y + \rho\,(A \cdot x^{+} - z^{+})
\end{aligned}
\tag{4.1}
$$

The following implementation details are noteworthy:

- The superscript $\cdot^{+}$ denotes the value at the next iteration.
- The vectors x, y, and z are represented either using fixed-point arithmetic or single precision floating point.
- The symbol $\cdot$ represents matrix multiplication.
- The penalty parameter ρ is chosen as a power of 2 to enable efficient bit-shift operations.
- $\Pi_{[\ell,u]}$ denotes clipping between the bounds ℓ and u, which can be optimized for efficient hardware implementation.
- All matrices are constant and hard-coded at bitstream compile time.

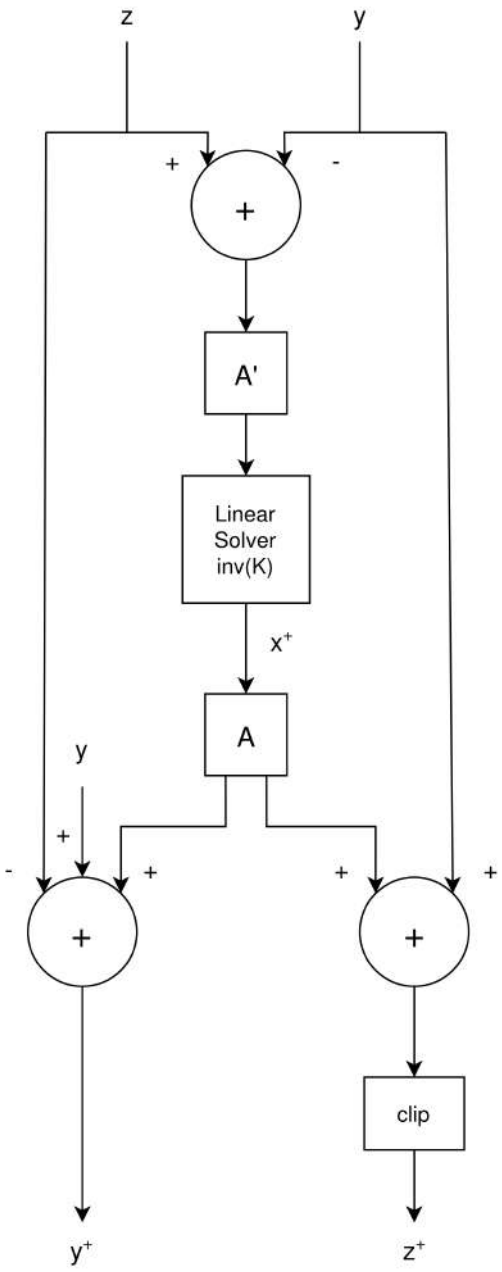


Figure 4.1: Dataflow diagram of the FPGA implementation.

4.3.1 Linear System Solving

The first step of the ADMM iteration (Eq. 4.1), which consists of solving a linear system, is the most computationally demanding part of the algorithm.

The coefficient matrix K is known in advance and exhibits a sparse, banded structure with low bandwidth (Fig. 4.2). This allows for an efficient, structure-exploiting linear solver.

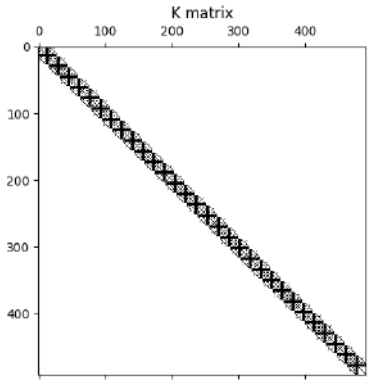


Figure 4.2: Sparsity pattern of the matrix K.

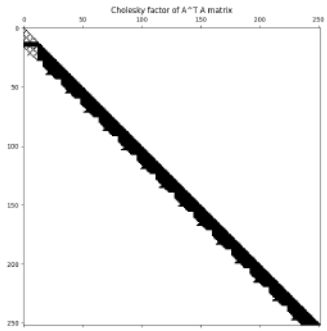


Figure 4.3: Sparsity pattern of the Cholesky factor L of K.

We perform a Cholesky factorization of the matrix:

$$K = LL^\top,$$

where L is a lower-triangular matrix that preserves much of the sparsity of K, as illustrated in Fig. 4.3.

Once the factorization is computed offline, the linear system $Kx = b$ can be solved online on the FPGA using a two-step procedure: first, forward substitution, followed by backward substitution.

For the current problem, the Cholesky factor L contains, on average, only 16 nonzero entries per row, with a maximum of 21. This sparsity pattern remains constant even as the problem size increases with a longer prediction horizon, enabling efficient hardware implementation.

Exploiting this structure allows for a highly efficient FPGA implementation that achieves real-time performance without excessive resource usage.

Relation to Riccati-Based MPC Solvers

Riccati-based MPC solvers such as TinyMPC [61] compute the solution of the linear system through a forward–backward recursion along the prediction horizon. Initially, this approach was avoided in the present implementation due to concerns about numerical error accumulation when using fixed-point arithmetic. However, triangular forward and backward substitution propagate numerical errors along the horizon in a similar manner. As a result, both approaches exhibit comparable numerical characteristics and identical asymptotic complexity. The main practical difference lies instead in the memory representation of the system matrices. TinyMPC employs a horizon-structured

representation that allows the same system matrices to be reused at each stage of the horizon, reducing the amount of stored data. In the implementation presented in this work, the matrices and factorizations are generated offline and stored explicitly, resulting in a memory footprint that grows more directly with the horizon length.

4.3.2 Efficient Storage of Matrices

As previously discussed, the implementation has been performed using HLS. The matrices required by the algorithm are generated offline using Python scripts, whose purpose is to produce header files containing all the hard-coded data. These headers are then compiled together with the algorithm into the FPGA design.

This allows for a division-free implementation of the algorithm, by leveraging the storage of reciprocal values, in order to only have multiplication operations.

A significant effort has been dedicated to optimizing the storage of these matrices, both in terms of memory footprint and access efficiency during the algorithm execution. Efficient storage is critical on the FPGA, where on-chip memory is limited and access patterns can significantly impact performance.

Cholesky Factor

The Cholesky factor $L \in \mathbb{R}^{n \times n}$ of the KKT matrix is stored using a banded-row format to exploit its sparsity. If the matrix has a bandwidth of h, it is compressed into an $n \times h$ form, denoted as L_{band}.

The compression can be visualized as follows:

$$L = \begin{bmatrix} l_{1,1} & 0 & 0 & \cdots & 0 \\ l_{2,1} & l_{2,2} & 0 & \cdots & 0 \\ l_{3,1} & l_{3,2} & l_{3,3} & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & l_{n,n} \end{bmatrix} \longrightarrow L_{\text{band}} = \begin{bmatrix} 0 & \cdots & 0 & l_{1,1}^{-1} \\ 0 & \cdots & l_{2,1} & l_{2,2}^{-1} \\ \vdots & \iddots & \vdots & \vdots \\ l_{n,n-h+1} & \cdots & l_{n,n-1} & l_{n,n}^{-1} \end{bmatrix}.$$

In L_{band}, each row contains only the nonzero elements within the bandwidth, with the **last element storing the reciprocal of the diagonal entry**. Zero-padding is applied at the beginning of the first few rows to maintain uniform row lengths.

This representation reduces both memory usage and computational overhead:

- Only the necessary band elements are stored.

- Access patterns are simple, enabling efficient forward and backward substitution on FPGA.
- Runtime divisions are eliminated by precomputing the reciprocals of the diagonal elements.

Constraint Matrix A

The equality and inequality constraint matrix $A \in \mathbb{R}^{p \times q}$ is stored in a *sparse row-wise format*. For each row, only the nonzero elements are retained, together with their corresponding column indices. This allows for compact representation of large, mostly-zero matrices and enables efficient memory access during matrix-vector multiplications in the ADMM iterations.

The compression of a single row of A can be visualized as follows:

$$\begin{aligned} \text{full row:} \quad & A_i = [0, 0, a_{i,3}, 0, a_{i,5}, 0, 0, a_{i,8}, 0] \\ \text{compressed:} \quad & \begin{bmatrix} a_{i,3} & a_{i,5} & a_{i,8} \end{bmatrix}, \\ \text{indices:} \quad & \begin{bmatrix} 3 & 5 & 8 \end{bmatrix}. \end{aligned} \tag{4.2}$$

By storing both the nonzero values and their column indices, we can:

- Reduce memory usage by omitting zeros.
- Keep a simple and regular access pattern for FPGA implementation.
- Compute the number of bits required for each index from the maximum column index, further minimizing memory requirements.

This sparse representation, together with the banded storage of the Cholesky factor, ensures that the ADMM solver can operate efficiently in real time on the resource-constrained Crazyflie hardware. It provides a good balance between memory efficiency and computational speed, allowing all linear algebra operations to be mapped naturally onto the FPGA.

4.3.3 Hardware-Oriented Implementation of the ADMM Solver

Once compact matrix storage formats are defined, the solver implementation is structured to efficiently map ADMM iterations onto FPGA hardware. Beyond reducing memory usage, a primary goal is to ensure predictable and pipeline-friendly execution patterns, minimizing costly memory accesses.

A key optimization enabling efficient implementation is the use of a **sliding window mechanism** during forward and backward substitution. In triangular system solving, each new variable depends

Algorithm 2 Forward Substitution with Sliding Window

Initialize window$[0 : h-2] \leftarrow 0$ ▷ Sliding window buffer
for $i = 0$ **to** $n-1$ **do**
    sum $\leftarrow 0$
    for $j = 0$ **to** $h-2$ **do** ▷ Dot product with window
        sum $\leftarrow$ sum $+ L[i, j] \cdot$ window$[j]$
    end for
    $x[i] \leftarrow (b[i] - \text{sum}) \cdot L[i, h-1]$ ▷ $L[i, h-1]$ stores L_{ii}^{-1}
    for $k = 0$ **to** $h-3$ **do** ▷ Shift window left
        window$[k] \leftarrow$ window$[k+1]$
    end for
    window$[h-2] \leftarrow x[i]$ ▷ Insert new value
end for

Algorithm 3 Sparse Matrix-Vector Multiplication ($A^T x$)

for $i = 0$ **to** $n-1$ **do**
    sum $\leftarrow 0$
    for $j = 0$ **to** nnz$[i] - 1$ **do** ▷ Only nonzero elements
        col_idx $\leftarrow$ indices$[i, j]$
        sum $\leftarrow$ sum $+ A^T_{\text{data}}[i, j] \cdot x[\text{col\_idx}]$
    end for
    $y[i] \leftarrow$ sum
end for

only on a limited number of previously computed values due to the banded structure of the Cholesky factor. Instead of repeatedly reading these values from memory, the solver keeps only the most recent elements in a small local buffer implemented as a shift register.

As computation progresses row by row, newly computed values are inserted into this window while older ones are shifted out. This sliding window always contains exactly the elements required for the next computation step. As a result, accesses to large on-chip memories are avoided, and all dependencies are satisfied using fast local registers.

This technique provides several important advantages for hardware implementation:

- Memory traffic is drastically reduced, since previously computed elements do not need to be reloaded.
- Access patterns become perfectly regular, which allows the HLS tool to pipeline loops efficiently.
- The required storage size depends only on the matrix bandwidth, not on the full problem dimension, allowing scalability to larger horizons without increasing local buffering costs.

The same principle applies to both forward and backward substitution, with the window shifting in opposite directions. Because each iteration reuses locally cached values, the linear system solve becomes a streaming computation with minimal latency between successive outputs.

The remaining parts of the ADMM iteration are similarly structured to maintain sequential and predictable memory access. Sparse matrix-vector multiplications process rows sequentially, multiplying only stored nonzero elements, enabling efficient scheduling of multiply–accumulate operations.

4.3.4 Computational Complexity

The computational cost of the proposed solver can be analyzed by examining the operations performed during a single ADMM iteration. Each iteration consists of four main steps: the solution of a linear system through forward and backward substitution, the multiplication by the constraint matrix A, the multiplication by its transpose $A^\top$, and the elementwise updates associated with the projection and dual-variable updates.

The linear system is solved using triangular forward and backward substitutions based on the precomputed banded Cholesky factors. Let h denote the bandwidth of the factor. Since only elements within the band are processed, each row update requires $\mathcal{O}(h)$ operations. The total cost of a triangular solve therefore becomes

$$\mathcal{O}(N_{\text{var}} h). \tag{4.3}$$

Because the MPC formulation only introduces couplings between variables belonging to the same stage or to two consecutive stages of the prediction horizon, the resulting KKT matrix has a block-banded structure whose bandwidth depends only on the state and input dimensions and therefore remains constant as the horizon length increases. It can therefore be treated as $h = \mathcal{O}(1)$, which leads to a cost

$$\mathcal{O}(N_{\text{var}}) \tag{4.4}$$

for both the forward and backward substitutions.

The multiplications with the sparse matrices A and $A^\top$ are implemented using compressed storage that contains only the nonzero elements. If the number of nonzero entries per row remains bounded independently of the horizon, each row requires $\mathcal{O}(1)$ work, leading to

$$Ax : \mathcal{O}(N_{\text{constr}}), \qquad A^\top y : \mathcal{O}(N_{\text{var}}). \tag{4.5}$$

The remaining operations in the ADMM iteration correspond to elementwise updates of the z and y variables, including projections onto box constraints and multiplications by powers of two used to implement the ρ parameter. These operations are performed independently for each constraint and therefore scale linearly with the number of constraints.

Combining the above contributions, the computational cost of one ADMM iteration is

$$\mathcal{O}(N_{\text{var}} + N_{\text{constr}}). \tag{4.6}$$

For a fixed state dimension, input dimension, and constraint structure, both N_{var} and N_{constr} grow linearly with the prediction horizon H. As a result, the cost of one ADMM iteration scales as

$$\mathcal{O}(H). \tag{4.7}$$

If k ADMM iterations are executed at each control step, the overall complexity of the solver becomes

$$\mathcal{O}(kH). \tag{4.8}$$

The memory complexity follows a similar trend. The banded factor stores only $\mathcal{O}(h)$ elements per row, and the sparse constraint matrices contain a constant number of nonzero entries per row. Consequently, the total storage required by the solver also scales linearly with the horizon:

$$\mathcal{O}(H). \tag{4.9}$$

4.3.5 Floating-Point vs Fixed-Point Datatypes

The solver is implemented in a datatype-agnostic manner using template-based definitions in the HLS code. As a result, the same implementation can be synthesized either with IEEE 754 single-precision floating-point arithmetic or with fixed-point arithmetic by modifying a single datatype definition.

In the fixed-point configuration the following datatype is used:

```
typedef ap_fixed<32,10, AP_RND, AP_SAT> fp_t;
```

This definition specifies a 32-bit fixed-point number with 10 integer bits (including the sign bit) and 22 fractional bits. The template parameters also define the rounding and overflow behavior. The `AP_RND` mode enables rounding to the nearest representable value during arithmetic operations, which improves numerical accuracy compared to simple truncation. The `AP_SAT` mode enables saturation on overflow, ensuring that values exceeding the representable range are clamped to the maximum or minimum representable value rather than wrapping around. This behavior improves numerical robustness and prevents instability due to arithmetic overflow.

Both floating-point and fixed-point variants of the solver were synthesized and evaluated. In terms of control performance, the difference between the two representations was negligible in simulation. However, the fixed-point implementation provides advantages in hardware efficiency, including reduced execution latency and lower energy consumption. For this reason, the fixed-point implementation is used in the flight experiments presented in the following chapters.

It is important to note that during HLS *software simulation* the floating-point version executes significantly faster than the fixed-point version. Floating-point arithmetic is executed directly by the host processor's hardware floating-point unit, while the `ap_fixed` datatype must emulate fixed-point arithmetic in software, including the rounding and saturation behavior specified by `AP_RND` and `AP_SAT`. As a result, HLS simulation of the fixed-point implementation is typically an order of magnitude slower than the floating-point version. This difference affects only the software simulation phase and does not reflect the performance of the synthesized hardware implementation.

4.3.6 Warm Starting of the Solver

The solver uses a simple warm-starting strategy in which all optimization variables are preserved between successive MPC solves. Specifically, the primal variables, dual variables, and Lagrange multipliers are maintained across control iterations and used as the initial point for the next ADMM solve.

More advanced warm-starting techniques commonly used in MPC involve shifting the predicted

trajectories forward in time in order to align the solution with the new optimization horizon. In the present implementation this strategy is not employed.

The controller operates at a significantly higher rate than the MPC prediction step. Consequently, the optimization problem changes only slightly between consecutive solver executions. Under these conditions, directly reusing the previous solution provides an effective initialization without requiring explicit horizon shifting.

4.3.7 Handling of the ρ Parameter

To simplify the hardware implementation, the ADMM penalty parameter ρ is constrained to be a power of two. This design choice enables efficient implementation for both fixed-point and floating-point arithmetic.

In the fixed-point implementation, scaling by ρ can be realized as a bit-shift operation, which avoids the need for a full multiplication unit. In the floating-point implementation, the scaling is performed using the `ldexp` function from the C++ standard library, which corresponds to exponent adjustment and synthesizes to simpler hardware than a general floating-point multiplier.

Following the approach used in `OSQP`, the implementation supports two distinct ρ values: one associated with equality constraints and one associated with inequality constraints. Per-variable penalty parameters are not supported. Restricting the solver to two global values allows efficient implementation while retaining sufficient flexibility for practical tuning.

4.3.8 Accumulator Data Width

Many operations in the solver consist of multiply–accumulate sequences, such as those arising in dot products and sparse matrix–vector multiplications. In the fixed-point implementation, these operations require careful selection of the accumulator register width in order to prevent numerical overflow.

To ensure safe accumulation, the accumulator is implemented with a wider bit width than the input operands. This additional precision allows intermediate sums of multiple products to be represented without overflow while maintaining the desired numerical accuracy of the solver.

4.3.9 Setpoint Feeding

The MPC solver is formulated using a delta-state representation. In this formulation, the optimization variables represent deviations from a reference state rather than absolute states. The reference

setpoint is therefore handled by the microcontroller, which computes the difference between the current state and the desired setpoint and sends this deviation to the FPGA solver.

A limitation of this approach is that the solver assumes the reference setpoint remains constant over the entire prediction horizon. This assumption is appropriate for point-to-point transitions, where the objective is to stabilize the system at a fixed target. However, when tracking dynamic trajectories, improved performance can be achieved by providing a time-varying reference sequence and allowing the solver to exploit trajectory preview.

To reduce communication overhead between the FPGA and the microcontroller, the system includes the option to store precomputed reference trajectories directly in the FPGA memory. In this configuration, the FPGA can access the reference sequence locally during the MPC optimization without requiring continuous updates from the microcontroller.

This approach introduces a trade-off between trajectory storage and MPC horizon length, as both consume the limited on-chip BRAM resources. In the current implementation, horizons of approximately 70–80 steps approach the available BRAM capacity. For typical horizons between 20 and 40 steps, it is possible to store trajectories containing up to approximately 3000 reference points within the FPGA memory.

4.4 FPGA Onboard: PCB Design

To enable fully onboard execution of the MPC controller, a custom printed circuit board (PCB) was designed to host the Artix-7 FPGA and interface directly with the Crazyflie quadrotor. The design was inspired by previous work on FPGA-based control for UAVs, in particular the neural thrust controller developed by Azem et al. [56], which demonstrated the feasibility of mounting an FPGA directly on the Crazyflie platform.

The PCB design also draws from a commercially available development board, the Digilent Arty A7, which features the same FPGA model.

A key design goal was to ensure that all components could be easily sourced and assembled through JLCPCB's PCB assembly service. This constrained the component selection to parts available in JLCPCB's component library, favoring commonly stocked ICs and passives. This choice significantly reduced manufacturing complexity and cost, enabling rapid prototyping.

The following subsections describe the key functional blocks of the schematic design, which is shown in Fig. 4.4.

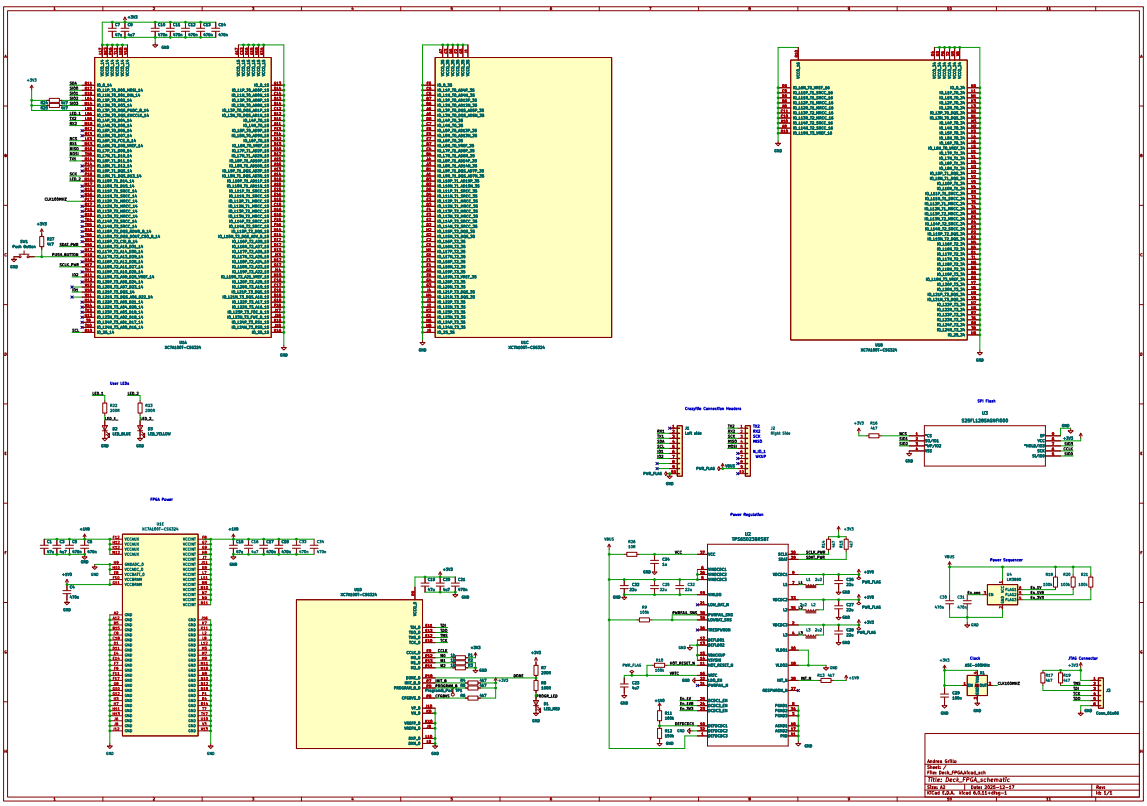


Figure 4.4: Schematic overview of the custom FPGA PCB.

The main sections of the schematic are as follows:

- **Power Regulation:** A *TPS65023* IC was selected to provide the three voltages required by the FPGA (1.0 V, 1.8 V, and 3.3 V). It can be powered directly from the Crazyflie's single-cell LiPo battery (3.3–4.2 V). A power sequencer, the LM3880, ensures proper FPGA power-up sequencing. The supporting circuitry, including inductors and capacitors, was inspired by the design from Azem et al.
- **SPI Flash:** A *S25FL128* SPI flash memory stores the FPGA bitstream and potentially additional user data.
- **Clock:** A 100 MHz oscillator, following the Arty A7 design, provides a fast clock to support the MPC algorithms.
- **JTAG Connection:** An external JTAG interface allows FPGA programming, SPI flash programming, and debugging.

- **Crazyflie Communication:** Multiple interfaces were included for flexibility: I^2C, two UARTs, SPI, and two GPIOs. While SPI is the planned primary communication method, the additional interfaces allow testing and future expansion.
- **User Push Button:** Directly connected to the FPGA, the button can trigger user-defined functions, such as resetting the algorithm.
- **LED Indicators:** Three LEDs provide status feedback: one connected to the FPGA `DONE` pin, and two general-purpose LEDs to signal algorithm stages.
- **FPGA I/O Bank Configuration:** Only the necessary I/O banks are powered, while all others are grounded, in accordance with Artix-7 documentation.

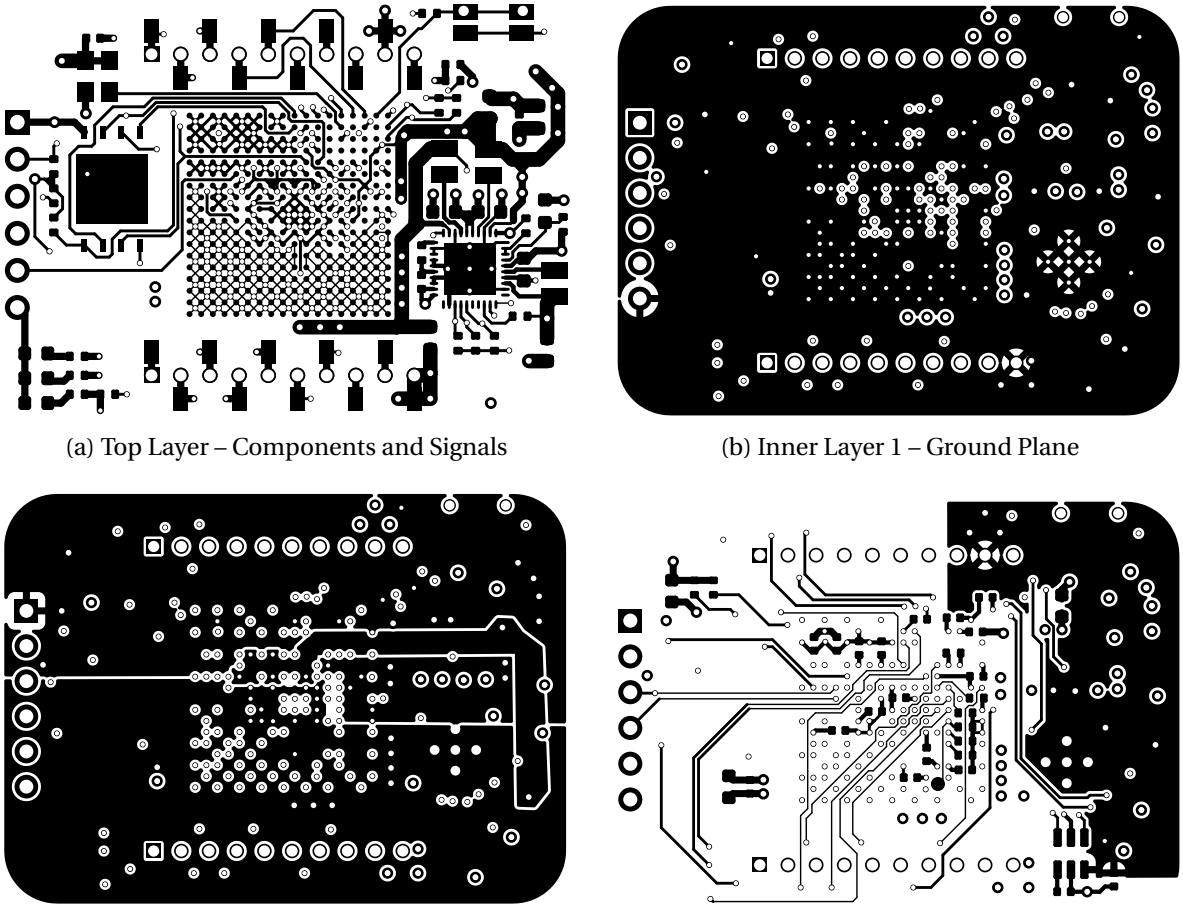
4.4.1 PCB Layer Stackup

The Artix-7 100T FPGA is packaged in a CSG324 ball grid array (BGA) with a 0.8 mm pitch, which presents significant routing challenges due to the high pin density. To accommodate the escape routing from the BGA and ensure signal integrity, a 4-layer PCB stackup was chosen.

The layer assignment is as follows:

- **Layer 1 (Top):** Signal routing and component placement.
- **Layer 2 (Inner 1):** Solid ground plane, providing a low-impedance return path and shielding.
- **Layer 3 (Inner 2):** Split power plane, carrying the multiple voltage rails required by the FPGA.
- **Layer 4 (Bottom):** Signal routing and additional component placement, mainly decoupling capacitors and resistors.

The use of dedicated internal planes for ground and power significantly improves power delivery integrity and reduces electromagnetic interference (EMI). The split power plane on Layer 3 is carefully partitioned to separate the core, auxiliary, and I/O voltage domains, with appropriate decoupling capacitors placed close to the FPGA power pins. The complete PCB layout for all four layers is shown in Fig. 4.5.



(a) Top Layer – Components and Signals

(b) Inner Layer 1 – Ground Plane

(c) Inner Layer 2 – Split Power Plane

(d) Bottom Layer – Signals

Figure 4.5: PCB layout of the custom FPGA board.

4.4.2 Power Supply and Sequencing

The Artix-7 FPGA requires multiple voltage rails with specific sequencing requirements. According to the Xilinx power guidelines, the core voltage (V_{CCINT} = 1.0 V) must be applied before or simultaneously with the auxiliary voltage (V_{CCAUX} = 1.8 V), which in turn must precede the I/O bank voltages (V_{CCIO}).

The power subsystem uses a combination of switching regulators and low-dropout (LDO) regulators to generate the required rails from the battery voltage supplied by the Crazyflie. A power sequencer IC ensures that the voltage rails are brought up in the correct order during power-on and are shut down in reverse order during power-off, preventing latch-up conditions and ensuring reliable FPGA initialization.

The power budget was estimated using Xilinx Vivado's power analysis tools, taking into account the

expected switching activity of the ADMM solver and the clock frequency. This analysis informed the selection of regulators with adequate current capacity and thermal performance.

4.4.3 Connection to Crazyflie

The PCB interfaces with the Crazyflie through its expansion deck connector, which provides access to power, ground, and several communication buses. The primary communication interface between the FPGA and the Crazyflie's STM32 microcontroller is SPI, which offers sufficient bandwidth for exchanging state estimates and control commands at the required control rate.

Additional GPIO lines are exposed for synchronization signals and debugging purposes. The mechanical design ensures that the PCB aligns with the Crazyflie's deck mounting system, maintaining the center of gravity close to the original position to minimize impact on flight dynamics.

4.4.4 Clock

The FPGA requires an external clock source to drive its internal logic. A low-jitter crystal oscillator is included on the PCB, providing a stable reference clock. The oscillator frequency was chosen to be compatible with the FPGA's clock management tiles, inspired by the commercial board Arty A7.

4.4.5 Programming Interface

The FPGA is programmed via a JTAG interface, which is exposed through a standard connector on the PCB. This allows the use of a Xilinx-compatible programmer (such as the Digilent JTAG-HS3) for bitstream loading and debugging. The JTAG interface also supports flashing the SPI flash.

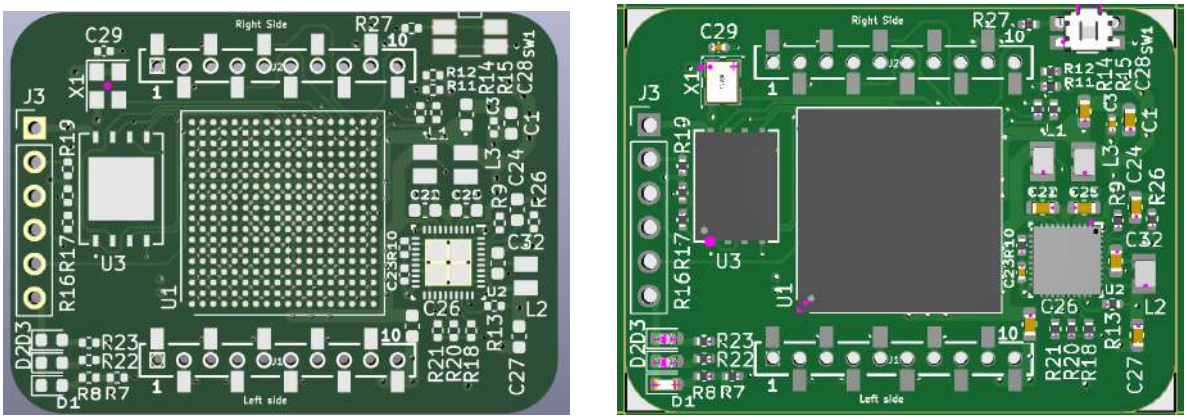
4.4.6 SPI Flash

To enable standalone operation without a connected programmer, the PCB includes an SPI flash memory. The FPGA is configured to load its bitstream from the flash at power-on, allowing the board to operate autonomously once programmed. The flash size was selected to accommodate the bitstream with margin for potential user data.

4.4.7 LEDs

Three LEDs are included on the PCB to provide visual feedback during operation. Two of them are directly connected to FPGA I/O pins and can be used to indicate status information such as

power-on, configuration complete, controller active, or error conditions. During development, the LEDs are useful for debugging timing and communication issues without requiring a logic analyzer. The third LED is connected to the DONE pin of the FPGA, showing when the chip has completed the bitstream loading.



(a) Render without components

(b) Render with components

Figure 4.6: Render of the custom PCB.

4.5 System Integration

4.5.1 Integrating the HLS IP in an FPGA Project

Vitis HLS generates a hardware module packaged as an IP core, which must then be integrated into a Vivado project. The Vivado project provides the surrounding logic required for communication with external components, integration with other board peripherals, configuration of the SPI flash memory, and assignment of FPGA pins connected to the various subsystems. It also implements communication interfaces such as UART or SPI.

The main components of the Vivado project are:

- **Constraint file**: specifies pin assignments, clock sources, and electrical constraints.
- **Top-level entity**: connects all modules together and implements the main finite state machine controlling system operation.
- **Communication modules**: implement external communication protocols, such as SPI or UART, enabling data exchange with the microcontroller and debugging tools.

The HLS-generated IP core is instantiated within this structure and connected to the communication and control logic to form the complete hardware system.

4.5.2 Automated Build and Deployment Pipeline

To increase reproducibility and simplify adaptation of the project to future applications, the entire codebase has been integrated into an automated build system based on Makefiles. This system orchestrates all stages of development and deployment.

The automated workflow includes:

- Generation of controller matrices from the Python-based system model.
- Simulation and co-simulation of the HLS code using Vitis HLS.
- Synthesis of the HLS code into a Verilog IP core.
- Synthesis and implementation of the complete Vivado FPGA project.
- Programming of the FPGA and flashing of the SPI memory.

An additional feature of the build system is the ability to execute computationally intensive steps on a remote server, followed by synchronization of generated artifacts to a local development machine for deployment. Automating these steps significantly accelerates development cycles and enables rapid experimentation and iteration.

4.5.3 SPI Communication

Several communication interfaces are available between the STM32F405 microcontroller and the Artix-7 FPGA on the board, including two UART ports, an I^2C bus, and an SPI interface. Since the objective is to achieve the highest possible control loop frequency, SPI was selected due to its high throughput and low protocol overhead. A UART interface was also implemented for early development and debugging purposes on evaluation hardware.

SPI communication relies on four main signals: MOSI and MISO for full-duplex data transfer, SCK as the clock signal, and a chip-select line used to address the slave device. Because rapid synchronization between the FPGA and microcontroller is required, an additional interrupt line is used. This line is asserted by the FPGA once control computation is completed, allowing the microcontroller to promptly retrieve the new control command.

The SPI interface operates at the maximum reliable frequency supported by both devices, ensuring that communication latency does not become a bottleneck in the control loop execution.

4.5.4 Crazyflie Firmware Integration

The Crazyflie firmware developed by Bitcraze runs on FreeRTOS, enabling structured task scheduling and efficient peripheral handling.

The firmware is highly modular and provides two relevant extension mechanisms: APIs for implementing custom deck drivers and APIs for integrating out-of-tree controllers.

In this work, both extension mechanisms are leveraged: a custom deck driver implements communication with the FPGA, while an out-of-tree controller integrates the FPGA-based control computation into the flight control stack.

SPI communication between the microcontroller and FPGA uses DMA transfers, allowing data exchange to occur without occupying CPU time, thereby preserving processing resources for other firmware tasks.

The control loop is synchronized with the FPGA execution. Since FPGA computation is faster than the firmware control loop period, the microcontroller performs a lightweight polling of the interrupt signal. In practice, the FPGA has already completed computation when the control task executes, so the interrupt is already asserted and polling introduces negligible overhead.

By setting the SPI at 12MHz clock, the whole transaction (sending current state and retrieving the control commands) takes around $100\mu s$.

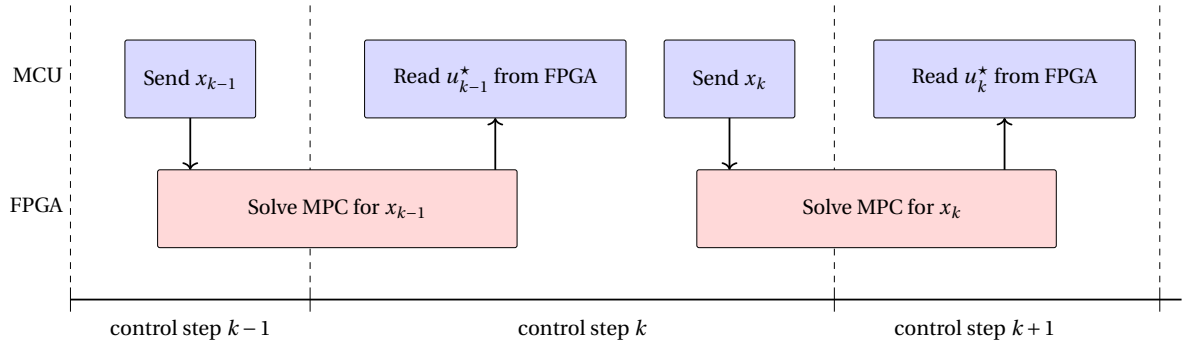
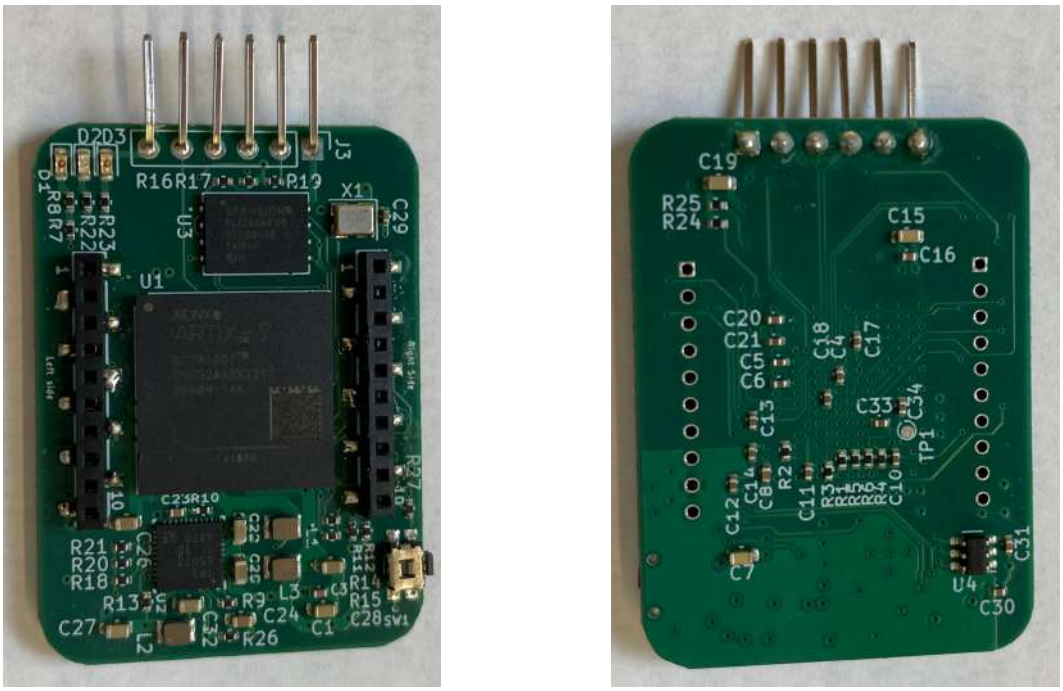


Figure 4.7: Illustration of the interaction between the MCU and the FPGA across consecutive control steps. During control step $k-1$, the MCU sends x_{k-1} to the FPGA. The corresponding MPC solve spans the interval between control steps $k-1$ and k. During control step k, the MCU reads the previously computed control input $u^\star_{k-1}$ and sends the new state x_k, which triggers the next FPGA solve. During control step $k+1$, the MCU reads the control input $u^\star_k$ computed during the previous interval.

4.5.5 Board Manufacturing and Drone Integration

The printed circuit boards were manufactured by JLCPCB [75]. All surface-mount components were assembled by the manufacturer. This was necessary because the FPGA package is a BGA device, which is difficult to solder reliably by hand. The only components assembled manually were the JTAG header and the two pin headers used to connect the board to the Crazyflie platform. The manufactured board is shown in Figure 4.8.



(a) Front (b) Back

Figure 4.8: Manufactured board.

The board was then integrated into the Crazyflie platform. As shown in Figure 4.9, the FPGA board is mounted beneath the main Crazyflie board, while the motion capture marker is mounted on top of the drone.

Figure 4.9: Drone with the FPGA board integrated beneath the main board and the motion capture marker mounted on top.

Chapter 5

Experimental Evaluation

5.1 Solver Benchmarking

In this section we evaluate the FPGA-based MPC controller.

The baseline chosen for comparison is TinyMPC, which is the state-of-the-art embedded MPC solver for the Crazyflie platform [61]. The FPGA accelerator achieves approximately 15× lower solve times and improves the energy–delay product by roughly 200× compared to the TinyMPC baseline, enabling substantially larger iteration budgets and longer prediction horizons within the same control period.

First, we evaluate the FPGA implementation in terms of latency, real-time feasibility, and energy–delay product across a range of operating points defined by the prediction horizon H and the ADMM iteration budget k.

Then, we describe the real-world experimental setup used for onboard deployment and explicitly delimit which closed-loop quantitative outcomes are included in this manuscript.

5.1.1 Latency and Timing Analysis

Benchmark Methodology

The objective of the benchmark is to compare solver latency and real-time feasibility for three implementations:

- **TinyMPC on Crazyflie MCU** (firmware baseline),

- **FPGA floating-point** implementation,
- **FPGA fixed-point** implementation.

All results use the same sweep:

$$H \in \{10,20,30,40,50,60,70,80\}, \quad k \in \{1,2,5,10,15,20,25,30\},$$

where H is the prediction horizon and k is the ADMM iteration budget.

TinyMPC Benchmark Inside the Real Firmware Stack

TinyMPC [61] timings are measured in Crazyflie firmware, in the same FreeRTOS environment where control normally runs. This means the solver executes concurrently with the rest of the flight stack (estimation, communication, safety supervision, logging, and other periodic tasks), rather than in an isolated bare-metal loop. For each (H,k) point, the benchmark logs the solve time in μs. Each TinyMPC point is aggregated over 80 samples, to account for variability in execution timing.

Timing-Guard Handling During Benchmark Acquisition

To acquire complete timing curves also in overload regions (large H, large k), some timing guards used in normal operation were temporarily disabled during benchmarking. Otherwise, those guards would stop the firmware execution before collecting representative over-budget measurements. This is important for interpreting the data: the reported curves are intended for performance characterization, not for a deploy-as-is safety configuration.

FPGA Benchmark Data and Projection

For FPGA runs, as the execution time is deterministic, only a single run is measured by means of a hardware counter on the FPGA. The data has been measured for $k = 10$. Additional points in k are generated with linear projection ($T \propto k$) from measured cycle counts. This is because there is no overhead and the execution time is fixed per iteration. The $k = 10$ rows are measured, while the rest of the iteration sweep is extrapolated.

Latency Scaling Across Solvers

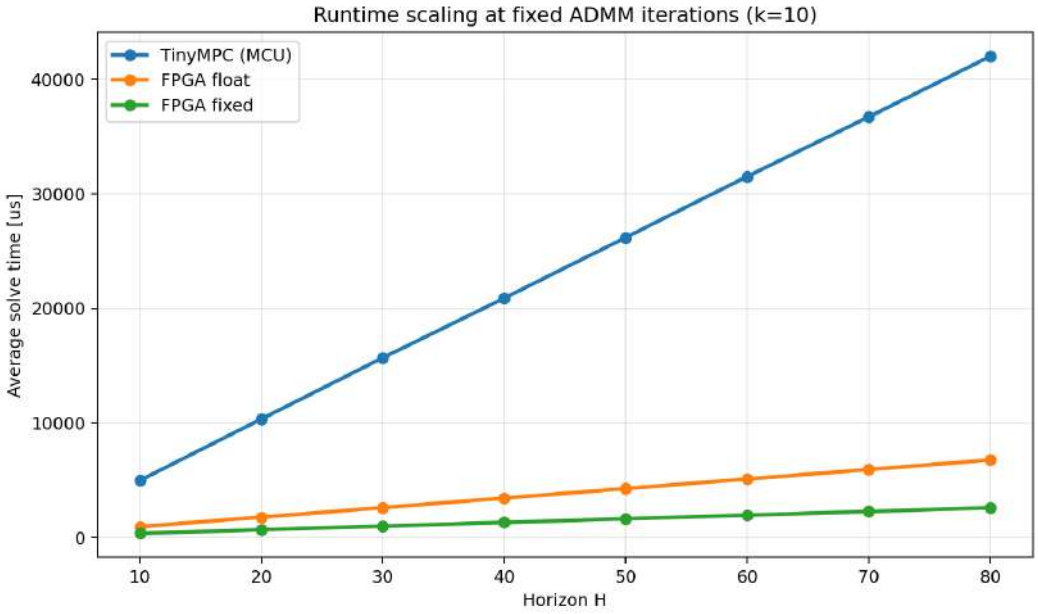


Figure 5.1: Average solve time versus horizon at fixed ADMM iterations ($k = 10$).

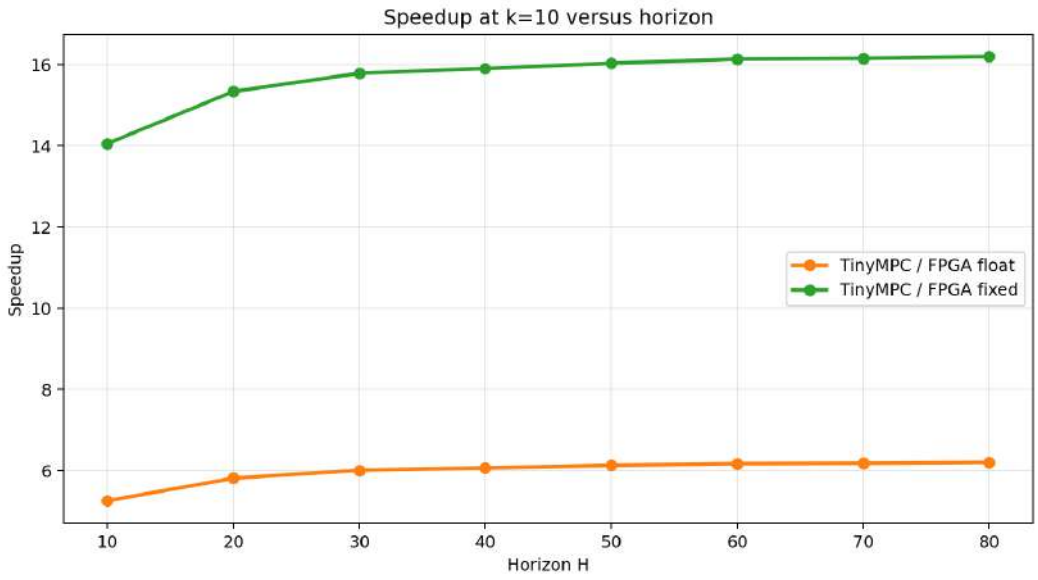


Figure 5.2: Speedup versus horizon at fixed ADMM iterations ($k = 10$).

Figure 5.1 provides a direct controlled comparison at $k = 10$: all implementations scale approximately linearly with horizon, but with clearly separated slopes. Figure 5.2 shows that this advantage remains nearly stable across prediction horizons. The speedup is lower for very short horizons, where the fixed communication overhead represents a larger fraction of the total execution time.

Median time per ADMM iteration per horizon step is:

- TinyMPC: 52.335 μs/(iter · step),
- FPGA floating-point: 8.577 μs/(iter · step),
- FPGA fixed-point: 3.277 μs/(iter · step).

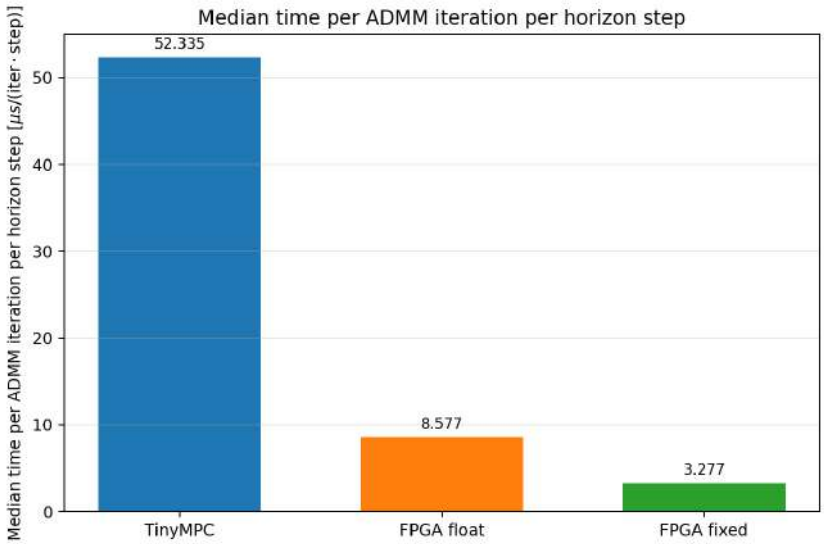


Figure 5.3: Median time per ADMM iteration per horizon step across solvers.

Thus, relative to TinyMPC, the median speedup is 6.10× for FPGA floating-point and 15.97× for FPGA fixed-point. On the same FPGA platform, fixed-point is 2.62× faster than floating-point. This is the central quantitative result: the hardware-algorithm co-design translates directly into order-of-magnitude latency reduction compared to software implementations.

Real-Time Feasibility at 500 Hz

For a 500 Hz control loop, the period is 2000 μs. For TinyMPC, feasibility is reported from measured sweep points. For FPGA, feasibility is estimated with an uncapped model $k_{\max} = \lfloor T_{\text{budget}} / T_{\text{iter}} \rfloor$, where T_{iter} is extracted from measured $k = 10$ data. In all FPGA numbers below, we explicitly subtract

100 μs communication overhead from the 2000 μs control period, i.e., FPGA compute budget is 1900 μs.

Table 5.1 reports the resulting maximum feasible ADMM iterations.

Source	H = 10	H = 20	H = 30	H = 40	H = 50	H = 60	H = 70	H = 80
TinyMPC (MCU)	2	1	1	-	-	-	-	-
FPGA floating-point	20	10	7	5	4	3	3	2
FPGA fixed-point	53	28	19	14	11	9	8	7

Table 5.1: Maximum feasible ADMM iterations within a 500 Hz loop period. FPGA entries include 100 μs SPI overhead (compute budget 1900 μs).

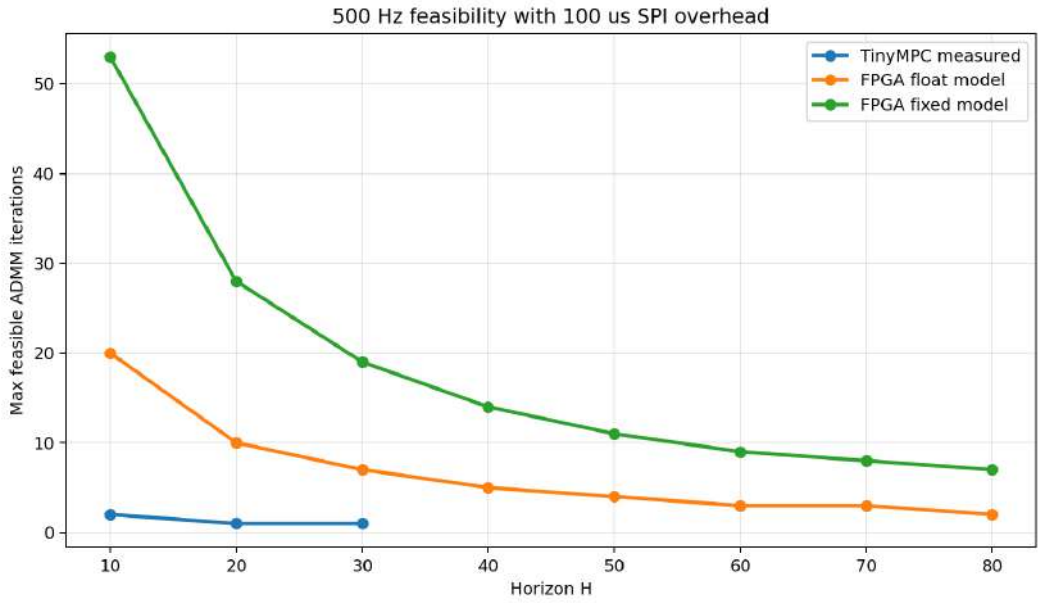


Figure 5.4: 500 Hz feasibility: maximum ADMM iterations versus horizon (TinyMPC measured, FPGA model-based, 100 μs SPI overhead included).

The key practical conclusion is that, at 500 Hz, TinyMPC on the MCU is constrained to very few iterations:

- at $H = 10$, at most $k = 2$,
- at $H = 20$ and $H = 30$, at most $k = 1$,
- for $H \geq 40$, no tested k fits the 2 ms budget.

This matches the observed operating regime: to maintain a 500 Hz loop on MCU, TinyMPC is effectively forced to run with 1–2 ADMM iterations in the low-horizon regime. Importantly, although $H = 30, k = 1$ is feasible on average (measured 1592 μs, max 1729 μs over 80 samples), TinyMPC runs under a preemptive FreeRTOS scheduler with concurrent estimator/radio/safety tasks, so timing is not perfectly deterministic. In practice, control deployment needs timing headroom, so configurations such as $H = 30, k = 1$ or $H = 20, k = 2$ is too close to the limit to be considered robust at 500 Hz. The configuration that is typically used by TinyMPC deployments is $H = 15, k = 2$, which is feasible but very limited in optimization budget.

In contrast, FPGA execution is cycle-deterministic: for a fixed configuration, computation latency is fixed. Even after including 100 μs communication overhead, FPGA acceleration enables much larger iteration budgets at the same loop rate (e.g., about $k = 19$ at $H = 30$ for fixed-point), allowing materially better optimization progress at each control update. In addition, moving the ADMM computation to FPGA recovers MCU time budget for the rest of the flight stack (state estimation, communication, safety checks, and logging), which is critical in real deployments.

Control-Rate Trade-Offs (100 / 250 / 500 Hz)

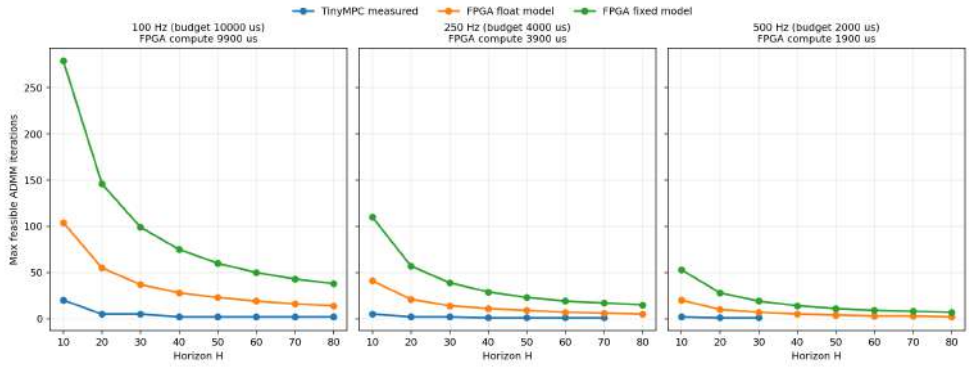


Figure 5.5: Maximum feasible ADMM iterations versus horizon for 100 Hz, 250 Hz, and 500 Hz (with 100 μs SPI overhead on FPGA).

Figure 5.5 makes the rate-accuracy trade-off explicit. TinyMPC can increase iterations at low rates, but its headroom collapses rapidly as the control rate increases. The FPGA implementations, especially fixed-point, retain high iteration budgets even in aggressive-rate regimes.

A complementary view, focused on feasible iterations at fixed horizon for specific control frequency, is shown in Figure 5.6 at $H = 20$. As the loop frequency increases from 100 Hz to 500 Hz, TinyMPC drops from $k_{\max} = 5$ to $k_{\max} = 1$, while FPGA float decreases from about 30 to 10, and FPGA fixed from about 30 to 25, preserving a much larger optimization budget at all rates. From a controller-design perspective, this directly expands the practically deployable operating region at high rates.

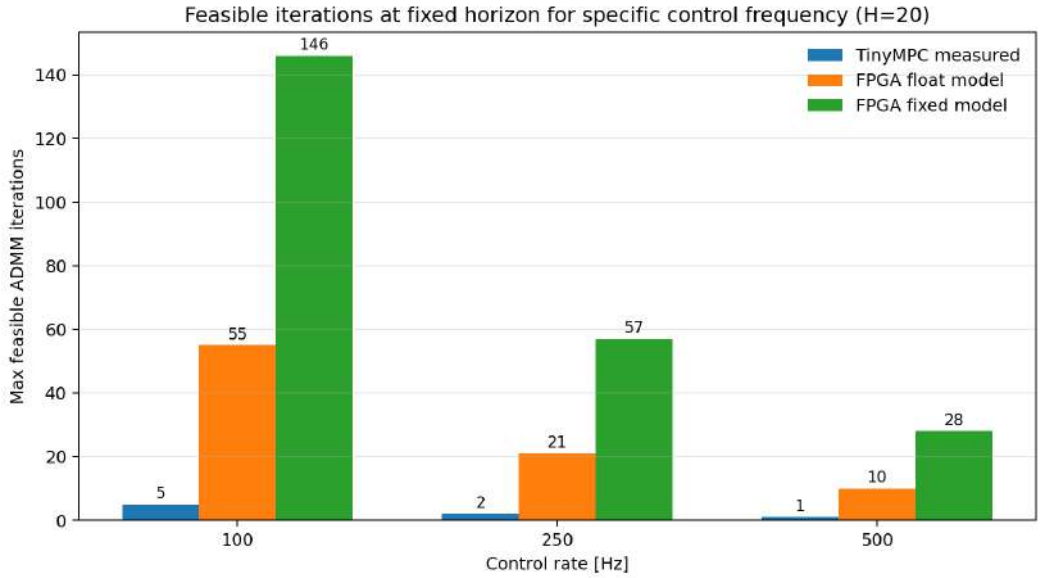


Figure 5.6: Feasible iterations at fixed horizon for specific control frequency ($H = 20$).

5.1.2 Resource Utilization Analysis

To connect runtime results with hardware cost, we analyze post-route utilization and power on the same (H, k) sweep. The main trend is clear: increasing horizon mostly stresses memory resources, while arithmetic resources remain comparatively stable. This is expected from the structure of the design, where longer horizons primarily increase the amount of stored problem data and intermediate state.

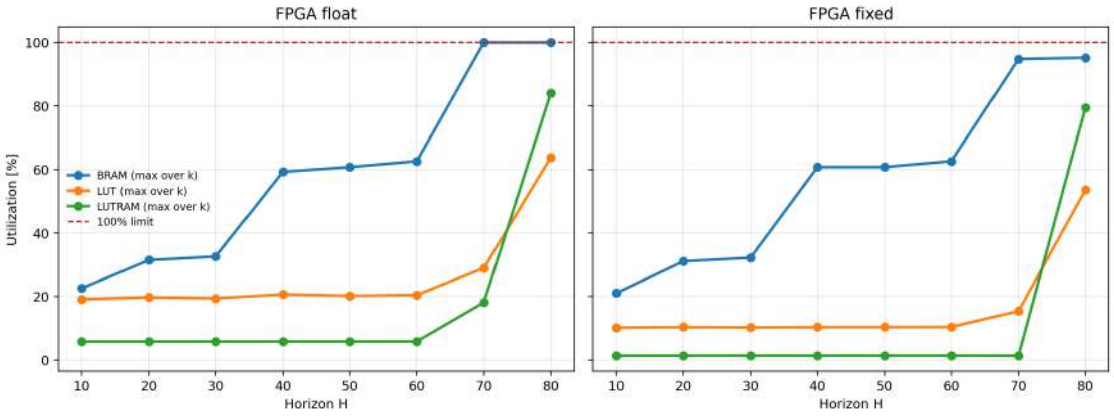


Figure 5.7: Resource utilization across horizon H, where each curve reports the maximum over the swept k values.

Figure 5.7 reports the utilization envelope versus horizon, where each metric is the maximum over all swept k at fixed H.

DSP utilization is almost flat across horizons (about 32.9% for float and 30.8% for fixed), and LUT usage is nearly constant until very large H. In contrast, BRAM usage increases with horizon. The increase is not perfectly smooth: plateaus such as $H = 20 \rightarrow 30$ and $H = 40 \rightarrow 60$ appear because BRAM is allocated in discrete physical blocks (RAMB18/RAMB36), not continuously. In practice, this means the tool often reserves the same number of BRAM blocks for multiple adjacent horizons, with partially filled blocks; usage jumps only when the next packing threshold is crossed.

This view makes the feasibility boundary explicit: BRAM reaches 100% in float at $H = 70, 80$, and once memory pressure becomes too high, LUT/LUTRAM maxima rise sharply. This is the key architectural transition at large H: when BRAM is no longer sufficient, storage is remapped into distributed RAM (LUTRAM), which explains the LUT explosion.

The maximum utilization across k is reported to provide a worst-case view of the resource envelope. It should be noted that the exact values depend on the Vivado synthesis and implementation process, whose optimizations are not fully deterministic; therefore, small variations in the reported utilization may occur across different runs.

5.1.3 Energy Consumption and Energy-Delay Product

After establishing where the design hits hardware limits, we now analyze energetic efficiency. This is important because, for embedded control, lower latency alone is not sufficient: the controller must also run within a tight power budget. To keep the comparison consistent with the timing study, we evaluate energy at the same operating points and use $k = 10$ as the reference slice across horizons.

For the FPGA implementations, energy is derived from post-route power analysis combined with measured solve time. For TinyMPC on the STM32F405, we use a datasheet-based estimate of active power: $I_{DD} = 87\,\text{mA}$ (typical run mode at 168 MHz, all peripherals enabled, $V_{DD} = 3.3\,\text{V}$, $T_A = 25^\circ\text{C}$), which corresponds to $P \approx 0.287\,\text{W}$ [76]. This provides a consistent baseline to compare architectural options even without direct board-level current instrumentation.

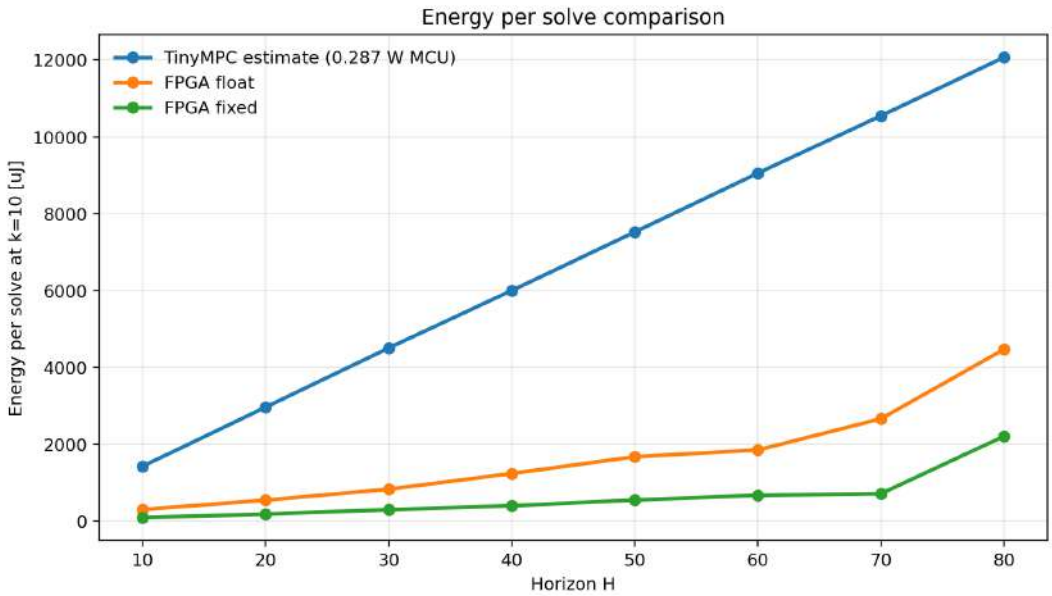


Figure 5.8: Energy per solve at $k = 10$: FPGA float, FPGA fixed, and TinyMPC estimate on STM32F405.

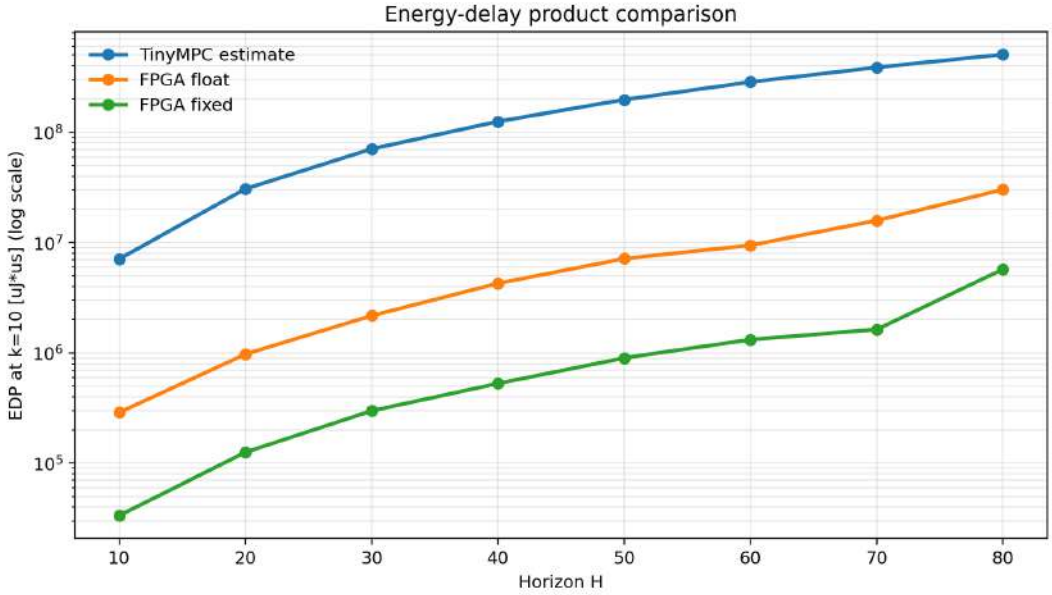


Figure 5.9: Energy-delay product (EDP) at $k = 10$, shown on a log scale.

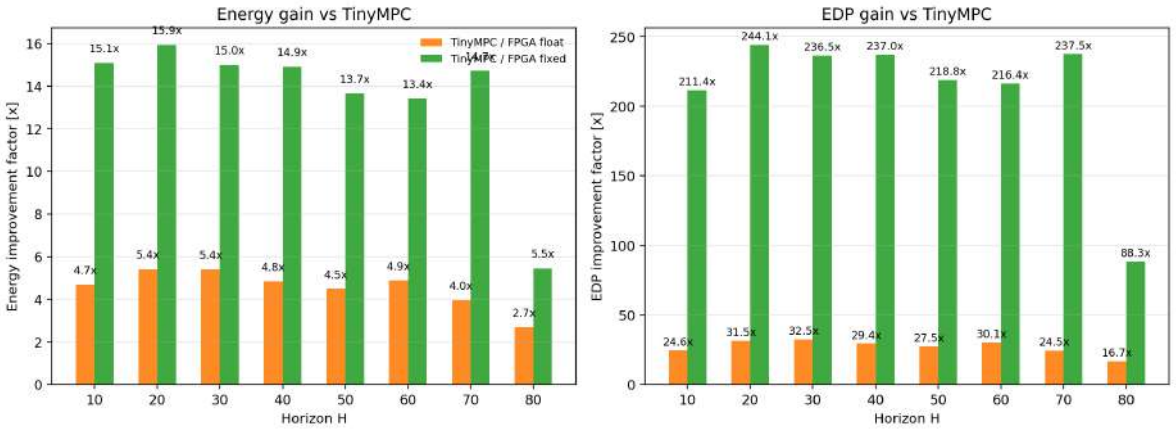


Figure 5.10: Per-horizon improvement factors (x) over TinyMPC at $k = 10$, for both energy and EDP.

Figure 5.8 shows that the fixed-point FPGA implementation consistently requires less energy per solve than the floating-point FPGA version, and both stay well below the TinyMPC estimate as horizon increases. Figure 5.9 extends this view with EDP, emphasizing not only how much energy is consumed, but how quickly useful work is completed.

The improvement-factor plot (Figure 5.10) summarizes this directly as speedup-like ratios in x: fixed-point FPGA reaches about 5.5× to 15.9× lower energy and 88× to 244× lower EDP than TinyMPC, while floating-point FPGA still improves over TinyMPC by about 2.7× to 5.4× in energy and 16.7× to 32.5× in EDP. The overall message is that fixed-point co-design improves both dimensions at once: it reduces execution time and also reduces energy, with the EDP metric showing the largest aggregate benefit.

5.2 Onboard Flight Experiments

This section presents two experimental demonstrations designed to evaluate the practical impact of the FPGA-accelerated MPC solver in closed-loop control. The experiments are conducted on the Crazyflie Brushless 2.1 quadcopter in an indoor motion-capture arena. In both experiments the control loop runs at 500 Hz, corresponding to a control period of 2 ms. The objective of these demonstrations is to assess how the computational capabilities of the FPGA implementation influence closed-loop control performance in realistic flight conditions.

5.2.1 Experimental Setup

All flight experiments are conducted in an indoor OptiTrack motion-capture environment equipped with eight cameras. The camera system measures the position of the vehicle and sends these

measurements to a motion-capture server. The server streams the data to the Crazyflie at 120 Hz. The same stream also provides the reference setpoint used by the controller.

All control computations are executed onboard the vehicle. Depending on the evaluated configuration, the optimization step is performed either by TinyMPC running on the microcontroller (MCU) or by the FPGA-based controller. In both configurations the state estimation pipeline remains on the MCU.

State estimation uses the Extended Kalman Filter (EKF) implemented in the standard Crazyflie firmware. The EKF fuses motion-capture measurements and inertial measurements from the onboard IMU. The mocap system provides absolute position measurements, while the IMU measurements are used to estimate attitude-related quantities.

In the current experimental setup the motion-capture system provides only position measurements and does not directly measure the orientation of the vehicle. This is due to the fact that only a single passive marker is mounted on the Crazyflie. Multi-marker configurations were tested, but the markers are very small and the spacing between them on this platform is limited, which resulted in unreliable tracking.

The architecture therefore keeps the control and optimization loop fully onboard. The offboard infrastructure provides the reference trajectory and the external position measurements, while state estimation and control-law computation remain on the vehicle.

Because orientation is not directly observed by the motion-capture system in this setup, attitude information is recovered through the fusion of IMU measurements within the EKF. This configuration reflects the practical sensing limitations of nano-quadrotor platforms and corresponds to a realistic onboard estimation scenario for experiments conducted on vehicles of this size.

All experiments presented are conducted using the fixed-point implementation of the FPGA solver.

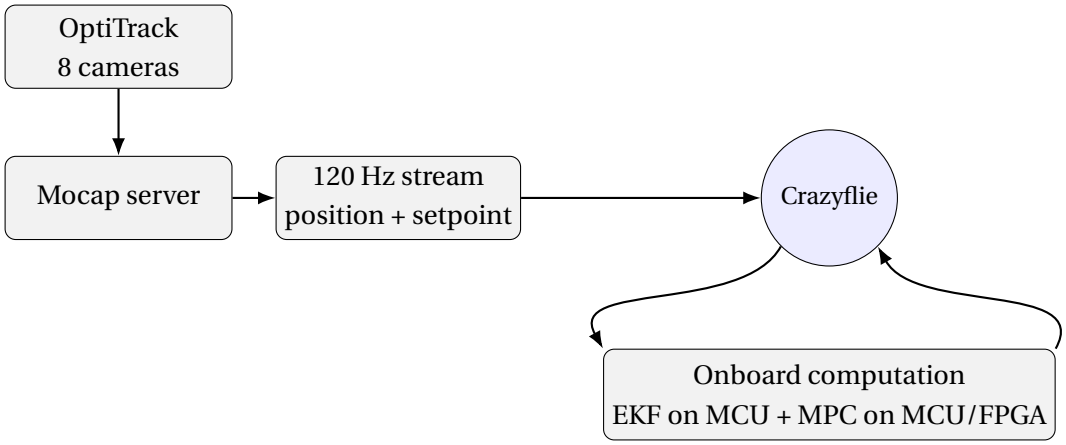


Figure 5.11: Experimental setup and data flow. OptiTrack measurements are processed by the mocap server and streamed at 120 Hz together with the reference setpoint. State estimation and control are executed onboard the Crazyflie.

5.2.2 Unconstrained Figure-Eight Trajectory Tracking

The first experiment evaluates trajectory tracking performance in the absence of state constraints. The objective is to compare the closed-loop behavior obtained using the FPGA-accelerated solver with that of the baseline TinyMPC implementation.

Both controllers operate at a sampling rate of 500 Hz. In the TinyMPC configuration the controller uses a prediction horizon of 15 steps and executes 2 ADMM iterations within the available 2 ms control period. In the FPGA configuration the controller uses a horizon of 20 steps and is able to execute 28 ADMM iterations within the same control period.

The reference trajectory consists of a smooth figure-eight path in the xy plane while maintaining a constant altitude. The maneuver requires continuous horizontal motion but does not approach actuator limits and does not impose additional constraints on the system.

Because no state constraints are present, the resulting quadratic program remains relatively simple. The optimization problem can therefore be solved within the control period both by the FPGA-accelerated implementation and by the MCU-based TinyMPC solver. In this regime the computational demand of the MPC problem is moderate, and both implementations are expected to achieve similar closed-loop performance.

Figure 5.12 shows the reference trajectory together with the executed trajectories obtained during the experiment. The left panel illustrates the vehicle motion in the xy plane and provides a comparison between the reference figure-eight trajectory and the measured trajectories for the two controller configurations. The right panel shows a long-exposure photograph of the Crazyflie during the experiment, where the LEDs of the drone trace the executed figure-eight path.

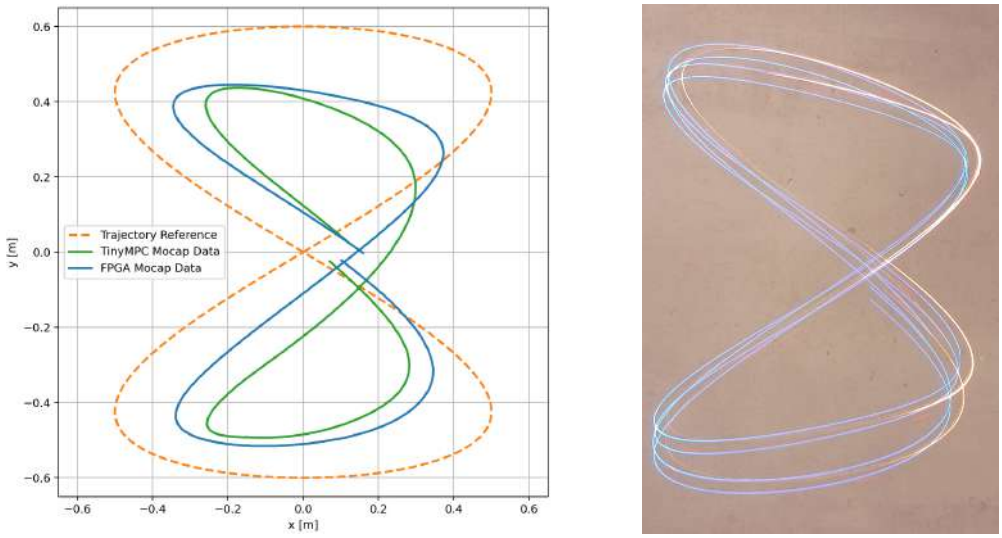


Figure 5.12: Unconstrained figure-eight trajectory tracking experiment. Left: overlay of the reference trajectory and the executed trajectories obtained with the TinyMPC and FPGA-based controllers in the xy plane. Right: long-exposure photograph of the Crazyflie executing the maneuver.

During the initial evaluation of this experiment it was observed that the primary limitation in tracking performance was not the optimization solver but model mismatch. To investigate this effect, the simulation model was extended to include additional physical phenomena such as aerodynamic drag and actuator motor lag. Incorporating these effects produced simulated trajectories that more closely resembled the behavior observed in the real system.

A series of parameter studies indicated that aerodynamic drag was the dominant source of the discrepancy, while the influence of other parameters was comparatively small. The quadrotor model used by the controller was therefore extended to include a drag term. After introducing this modification the trajectory tracking performance improved significantly.

Figure 5.13 shows the results obtained after incorporating aerodynamic drag into the system model. The left panel reports the motion capture trajectories projected in the xy plane together with the reference figure-eight trajectory. The improvement with respect to the previous model is visible for both implementations. The right panel shows a long-exposure photograph of the Crazyflie executing the same maneuver with the updated model.

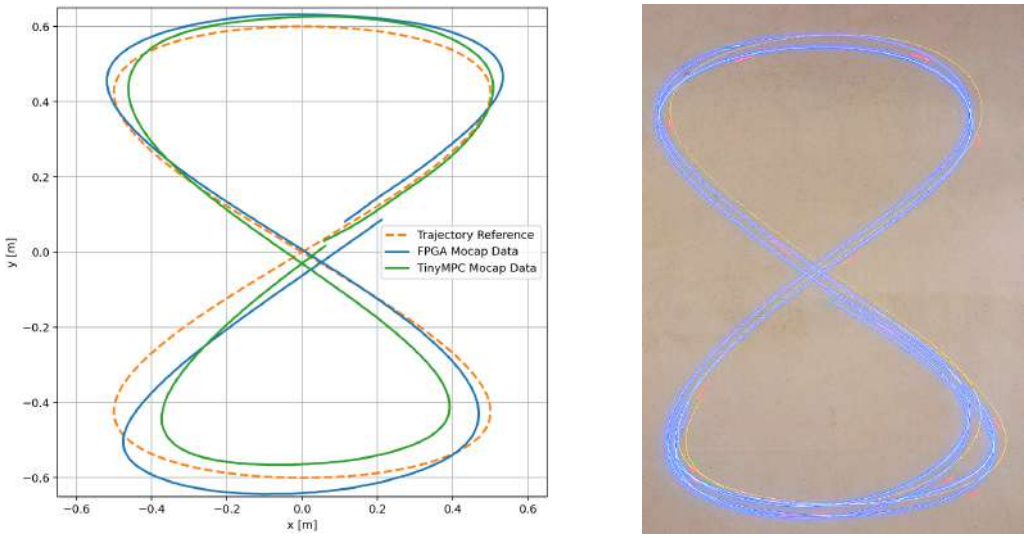


Figure 5.13: Unconstrained figure-eight trajectory tracking after incorporating aerodynamic drag in the model. Left: overlay of the reference trajectory and the executed trajectories obtained with the TinyMPC and FPGA-based controllers in the xy plane. Right: long-exposure photograph of the Crazyflie executing the maneuver.

These results confirm that when the optimization problem is relatively small, solver execution time is not the primary factor limiting control performance. In this regime the dominant limitations arise from modeling inaccuracies and from the linear approximation used by the controller.

Introducing an aerodynamic drag term improved the tracking performance for both the TinyMPC and FPGA implementations. This observation indicates that the dominant source of error in this scenario is model mismatch rather than solver performance.

As a result, both controller configurations achieve comparable closed-loop behavior for this unconstrained figure-eight tracking task. In the next experiment we consider a more demanding scenario in which the MPC controller must actively enforce state constraints during flight.

5.2.3 Constrained Trajectory Tracking

The second experiment evaluates the controller in a scenario that includes state constraints in the horizontal plane. Box constraints are imposed on the x and y coordinates of the vehicle, restricting the admissible motion to remain within a predefined square region.

The reference trajectory intentionally violates these bounds by following a star-shaped path centered inside the constrained region. As a result, the controller must modify the trajectory to maintain feasibility while still attempting to follow the reference motion.

Compared to the unconstrained case, this scenario significantly increases the computational requirements of the MPC problem. Handling state constraints effectively requires both a sufficiently long prediction horizon and a sufficient number of solver iterations at each control step in order to enforce feasibility while maintaining good tracking performance.

When constraints become active, the convergence of the ADMM algorithm typically requires additional iterations to satisfy the primal feasibility conditions associated with the constraint set. In practice, this increases the computational demand of the optimization step compared to unconstrained operation.

As discussed in the benchmark results presented in Section 5.1, the TinyMPC [61] implementation can execute only two ADMM iterations within the 2 ms control period required for a 500 Hz control loop on the Crazyflie MCU. With only two iterations available per control step, solver progress becomes limited when constraints are active. As a result, convergence toward the feasible set can be insufficient at a control rate of 500 Hz, making reliable enforcement of state constraints difficult under these timing constraints. For this reason, prior experimental studies using TinyMPC typically operate the controller at significantly lower control rates, often around 100 Hz or lower, in order to allow a larger number of solver iterations per control step and achieve adequate convergence of the optimization algorithm.

In contrast, the FPGA-accelerated implementation developed in this work is able to execute a substantially larger number of ADMM iterations within the same 2 ms control period. This allows constrained MPC problems to be addressed at a control rate of 500 Hz, providing sufficient solver progress at each control step to reliably enforce the state constraints while maintaining good tracking performance.

In this experiment, the controller is configured with a prediction horizon of 40 steps and executes 10 ADMM iterations at each control step.

Before executing the experiment on the real platform, the controller was evaluated in closed-loop simulation using the Vitis HLS simulation environment. This simulation reproduces the same solver implementation and numerical precision used in the FPGA design, allowing the expected controller behavior to be verified prior to deployment.

Figure 5.15 shows the reference trajectory together with the admissible region and the trajectory obtained in the closed-loop simulation. The admissible square region is indicated in the plot, while the reference trajectory intentionally leaves this region. The executed trajectory remains within the allowed area, demonstrating that the controller is able to enforce the state constraints in real time.

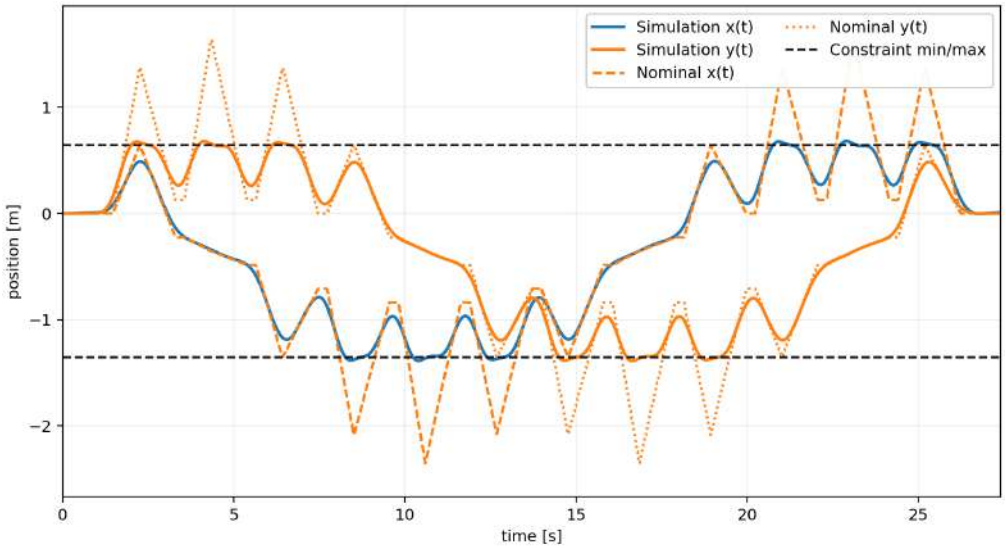


Figure 5.14: Position over time during the constrained trajectory tracking simulation.

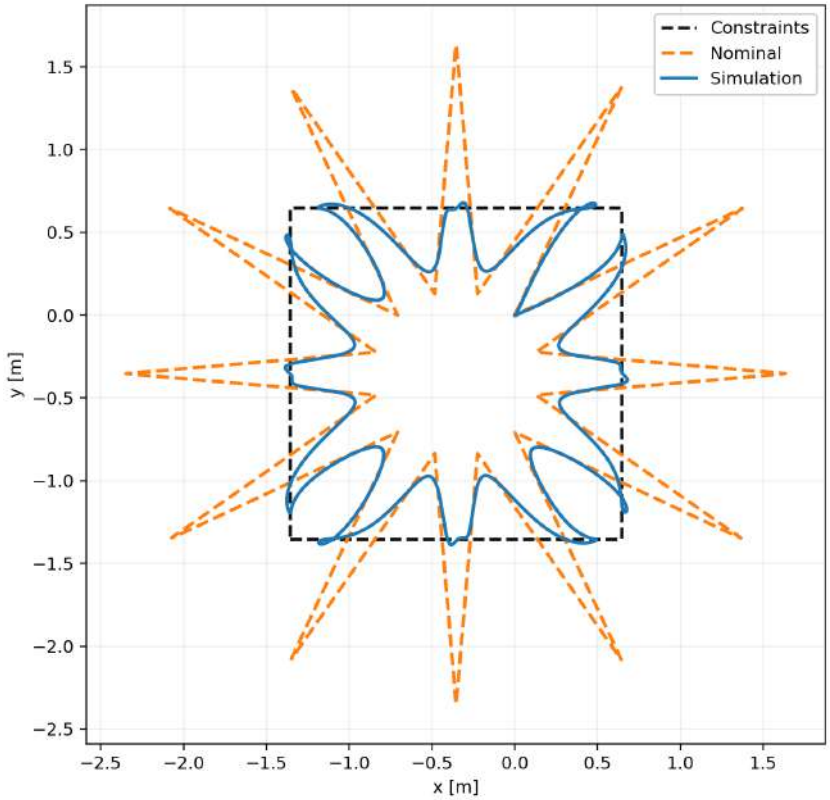


Figure 5.15: Constrained trajectory tracking experiment in simulation. The figure shows the trajectory in the xy plane together with the admissible square region.

The controller was then deployed on the real platform. Figure 5.16 shows a photograph of the Crazyflie executing the constrained trajectory tracking experiment. The image overlays the motion capture trajectory, the reference trajectory, and a composite visualization obtained by superimposing multiple frames of the flight video.

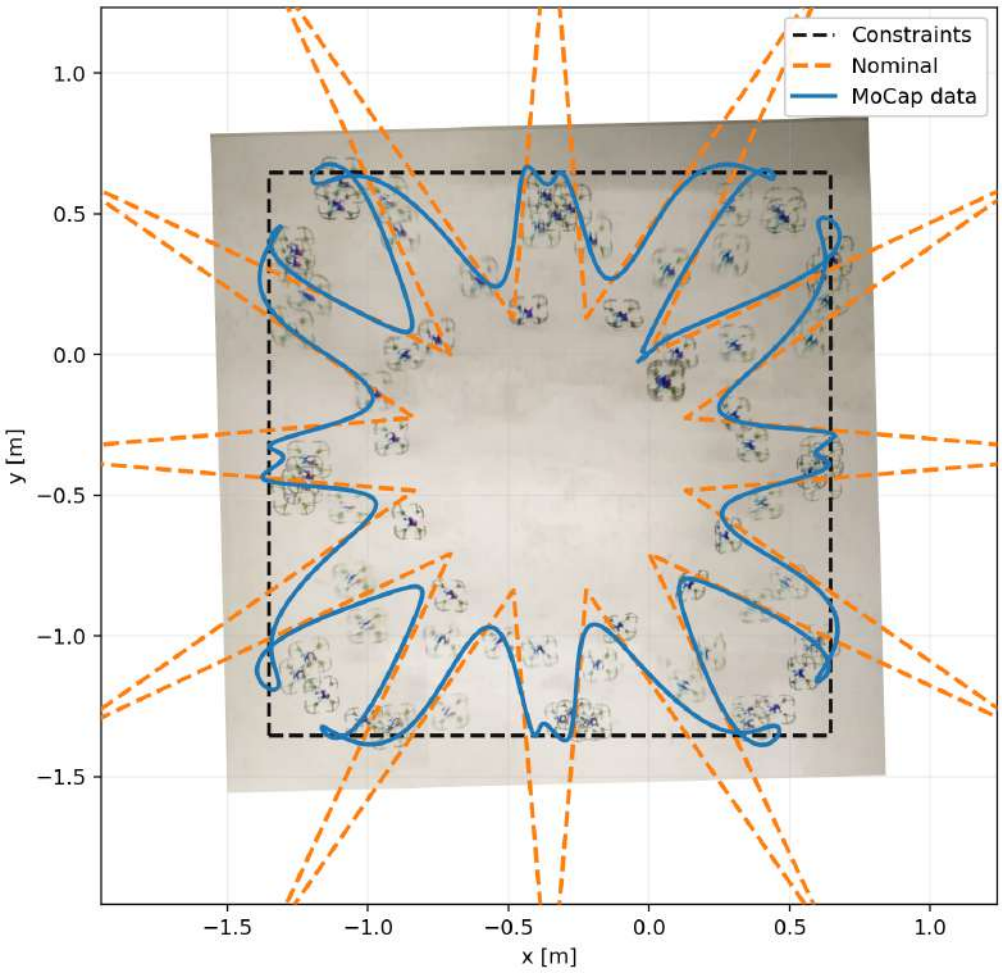


Figure 5.16: Photograph of the Crazyflie executing the constrained trajectory tracking experiment, overlaid with the reference trajectory and the box constraint.

The slight misalignment visible in the overlay is caused by camera distortion. The most accurate representation of the trajectory is provided by the motion capture data, shown in blue.

This experiment demonstrates that the FPGA-accelerated solver enables the execution of constrained MPC with longer horizons and sufficient solver iterations at a control rate of 500 Hz on a resource-constrained aerial platform. The controller is able to enforce the imposed state constraints while maintaining stable and smooth flight behavior throughout the experiment.

Chapter 6

Exploratory Study: Optimization on Neuromorphic Hardware

6.1 Motivation and Scope

Neuromorphic computing has recently emerged as a promising paradigm for energy-efficient computation, particularly for applications deployed at the edge where both power and computational resources are limited [38, 39]. Unlike conventional von Neumann architectures, which separate memory and computation and therefore suffer from the memory bandwidth bottleneck, neuromorphic processors integrate computation and memory within distributed networks of artificial neurons and synapses. This architecture enables event-driven computation and massive parallelism, which can lead to improved energy efficiency for certain classes of algorithms.

These characteristics are particularly appealing for embedded robotics. Small robotic platforms, such as micro aerial vehicles, operate under strict constraints on power consumption, computation, and memory bandwidth. As a result, deploying computationally demanding algorithms such as model predictive control (MPC) directly onboard these platforms remains challenging.

Recent advances in neuromorphic processors have opened the possibility of executing more complex algorithms on highly energy-efficient hardware. Modern systems such as Intel's Loihi 2 platform demonstrate that large networks of programmable spiking neurons can be implemented directly in silicon while maintaining low power consumption [39]. These architectures have primarily been developed for artificial intelligence applications, but their computational model may also be relevant for optimization-based control.

Several recent works have begun to investigate the use of neuromorphic systems for control and optimization tasks. Neuromorphic controllers based on spiking neural networks have been proposed

for robotic motion control, demonstrating that event-driven neural architectures can implement control laws with low energy consumption and real-time operation [77]. In addition, neuromorphic formulations of optimization algorithms have been explored for solving quadratic programs arising in MPC. For example, Mangalore et al. [57] propose a neuromorphic implementation of quadratic programming on Intel's Loihi 2 platform and report improvements in energy-delay product compared to conventional CPU and GPU solvers. Other works investigate the integration of spiking neural networks within adaptive or nonlinear MPC frameworks [78].

Despite these promising developments, the use of neuromorphic computing for real-time robotic control remains largely exploratory. Existing studies typically focus on algorithmic feasibility or simulation environments, and only limited work has investigated the integration of neuromorphic optimization algorithms within tightly constrained robotic platforms.

The goal of this chapter is therefore not to present a complete neuromorphic implementation of MPC, but rather to investigate the feasibility of mapping optimization algorithms for linear MPC onto neuromorphic hardware. In particular, this chapter addresses three questions:

- Which computational structures arising in MPC are compatible with neuromorphic architectures such as Loihi 2?
- How do existing neuromorphic optimization formulations relate to the structured optimization problems encountered in MPC?
- Can the numerical constraints of neuromorphic hardware, in particular its quantized weight representation, preserve the convergence behavior of iterative optimization algorithms?

The chapter therefore presents an exploratory study combining algorithmic analysis and preliminary numerical experiments.

6.2 Overview of Loihi 2

Neuromorphic computing aims to emulate key computational principles of biological neural systems. Unlike traditional von Neumann architectures, which separate memory and processing units, neuromorphic systems co-locate memory and computation, enabling low-latency and energy-efficient operations. Intel's Loihi 2 processor is a neuromorphic platform designed to accelerate spiking neural networks and to explore these computational principles in hardware.

In Loihi 2, information is transmitted through discrete events known as spikes rather than continuous numerical values. Computation occurs when neurons receive spikes from other neurons, making the execution model event-driven. In addition, Loihi 2 operates asynchronously without

a global clock, allowing neurons to update their states independently. These properties enable highly parallel execution and can reduce energy consumption in workloads characterized by sparse activity.

Another important architectural feature is the integration of memory and computation within each neuron and synapse. Each neuron maintains its internal state locally, and synaptic weights are stored nearby. This reduces the need for repeated transfers of data between memory and processing units, which is a major source of energy consumption in conventional processors.

6.2.1 Core Architecture

The Loihi 2 processor is composed of 128 neuromorphic cores interconnected through a network-on-chip that routes spike events between cores. Each core can host up to 1024 neurons together with their associated synaptic connections.

Each neuron maintains internal state variables such as membrane potential, refractory state, and synaptic traces. When a neuron receives spikes from presynaptic neurons, these events modify its internal state according to programmable dynamics. If the state crosses a specified threshold, the neuron generates an output spike that is routed to other neurons through the network-on-chip.

Synaptic connections carry spike events with programmable weights. Because computation occurs locally at the receiving neuron, this architecture enables large-scale parallelism across the chip.

Several architectural characteristics are particularly relevant when considering the implementation of numerical algorithms:

- **Event-driven execution:** computation occurs only in response to spikes.
- **Local memory access:** neuron states and synaptic weights are stored close to the computation.
- **Asynchronous updates:** neurons update independently rather than under a global clock.
- **Scalable communication:** spikes are routed across cores through the network-on-chip.

These characteristics suggest that algorithms based on sparse connectivity and local interactions may be particularly well suited for neuromorphic implementations [79].

6.2.2 Data Representation on Loihi 2

A key challenge when working with Loihi 2 is its unconventional data representation. Unlike standard CPUs and GPUs that use floating-point arithmetic with uniform precision, Loihi 2 relies on

fixed-point arithmetic and discrete spike events. This approach allows for highly energy-efficient computation but requires careful design to preserve numerical accuracy.

The main components of Loihi 2 data representation are:

- **Neuron State:** Each neuron maintains an internal state stored as a 24-bit signed integer. This internal state can include:
  - Membrane potential: representing the current activation of the neuron.
  - Refractory state: indicating if the neuron is temporarily unable to spike.
  - Synaptic traces: capturing temporal information for learning or dynamics.

  The 24-bit signed range, $[-2^{23}, 2^{23}-1]$, provides sufficient precision for most neural computations while keeping memory usage low. By maintaining local state, Loihi 2 reduces the need for frequent access to external memory, which improves both latency and energy efficiency.

- **Synaptic Weights:** Weights are stored using a mantissa-exponent format:
  - Mantissa: 8-bit signed integer, range $[-2^7, 2^7-1]$.
  - Exponent: 6-bit signed integer, range $[-2^5, 2^5-1]$. Each layer may use a global exponent or up to 16 different exponent groups.

  The effective weight is computed as
  $$w = m \cdot 2^e,$$
  where m is the mantissa and e is the exponent. This representation allows a wide dynamic range while keeping hardware resource usage compact. Using exponent groups enables different parts of a matrix to have different scaling factors, which is particularly useful for matrices with large dynamic ranges.

- **Spikes:** Neurons communicate through spikes, which can be represented using 1, 8, 16, 24, or 32 bits. The spike magnitude determines the impact on postsynaptic neurons. Larger bit-widths allow for higher precision but require more memory and communication bandwidth. Event-driven computation ensures that neurons update only when spikes occur, minimizing unnecessary energy usage.

- **Example – Matrix-Vector Multiplication:** Consider a layer performing
  $$y = Ax,$$
  where A is a synaptic weight matrix and x is a vector of spikes from the previous layer. If A has a large dynamic range, it can be partitioned into up to 16 exponent groups. The input vector x may consist of spikes with up to 32-bit precision. Using the mantissa-exponent format for weights and integer spikes for inputs, Loihi 2 can perform high-dynamic-range operations

efficiently without floating-point arithmetic. This motivates algorithmic designs that carefully consider precision limits, particularly for optimization-based tasks such as linear MPC.

While this representation enables efficient execution, it also introduces quantization constraints that must be considered when implementing numerical algorithms.

6.2.3 Programmable Microcode

Each neuron on Loihi 2 can execute a small program known as microcode. This program operates on the neuron's internal state and incoming spikes before generating output spikes. Microcode can therefore implement simple arithmetic operations, thresholding logic, or conditional updates.

Although the functionality of microcode is limited, it effectively transforms each neuron into a small event-driven processing element. Larger computations can then be constructed through networks of interacting neurons.

6.3 Neuromorphic Formulations of Quadratic Programming

One approach to implementing quadratic programming on neuromorphic hardware is the ReLU-QP formulation [80].

Because the ADMM updates consist of affine operations involving a linear system solve and vector additions, the entire iteration can be expressed as a single affine transformation. By precomputing the inverse of the KKT system offline, the iteration can be written as a matrix–vector multiplication followed by a projection.

The iteration can be written as

$$\begin{bmatrix} y^+ \\ z^+ \\ \lambda^+ \end{bmatrix} = \Pi_{(l,u)} \left(W_\rho \begin{bmatrix} y \\ z \\ \lambda \end{bmatrix} \right), \tag{6.1}$$

where W_ρ is a matrix precomputed offline and stored in memory. At runtime, each ADMM iteration therefore consists of a matrix–vector multiplication followed by a projection step enforcing box constraints.

This formulation is attractive for neuromorphic hardware because matrix–vector multiplication can be implemented naturally using neural network structures. A simplified conceptual architecture corresponding to this approach is illustrated in Figure 6.1.

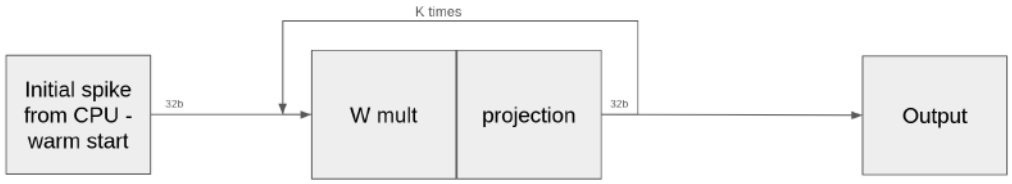


Figure 6.1: Conceptual architecture implementing ADMM iterations using a single linear transformation followed by a projection.

The high degree of parallelism available in neuromorphic systems also enables more complex architectures. For example, several ADMM pipelines could run in parallel with different values of the penalty parameter ρ. After a fixed number of iterations, a residual evaluation module could select the pipeline producing the best solution estimate. Figure 6.2 illustrates such a conceptual architecture.

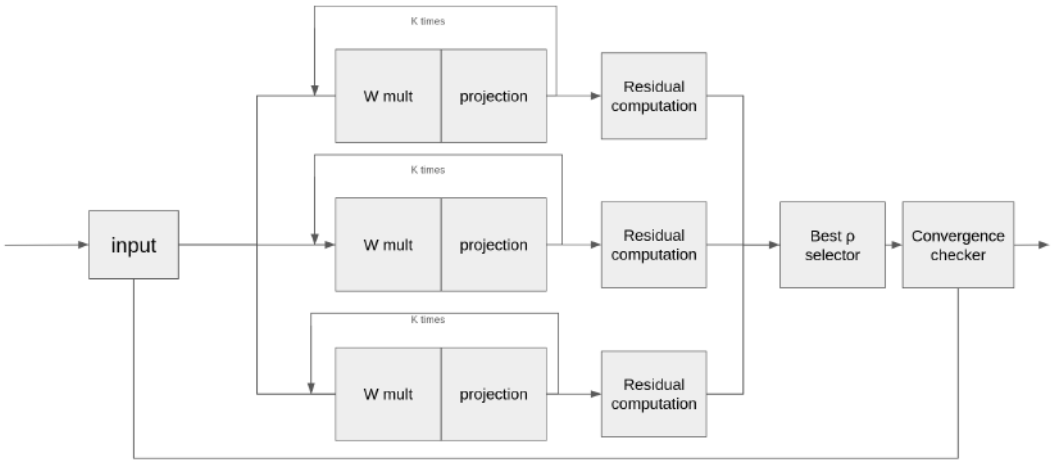


Figure 6.2: Conceptual parallel architecture exploring multiple ADMM parameter configurations.

While this formulation is appealing from an implementation perspective, it presents important limitations when applied to the structured quadratic programs arising in MPC. In particular, the matrix W_ρ must contain the inverse of the KKT matrix associated with the optimization problem. Although the KKT matrix itself typically exhibits a banded sparsity structure in MPC problems, its inverse is generally dense.

In practice, the inverse matrix contains very few zero entries even when the original KKT matrix is highly sparse. As a result, the corresponding neural network connectivity becomes dense. Moreover, the inverse of the KKT matrix appears repeatedly in the construction of the larger matrix, as illustrated in Figure 6.3, which shows the resulting sparsity pattern.

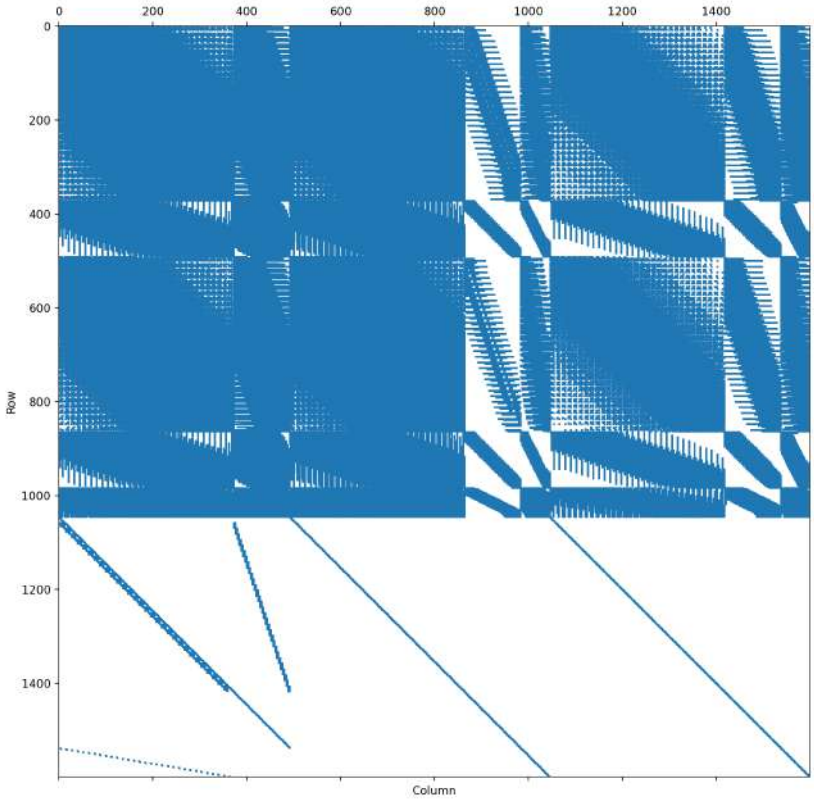


Figure 6.3: Sparsity pattern of the matrix W_ρ.

Neuromorphic architectures derive much of their efficiency from sparse communication and event-driven execution. In spiking neural networks, neurons transmit information only when spikes occur, and the associated communication cost is proportional to the number of active synaptic connections. When connectivity patterns are sparse, only a limited subset of neurons and synapses participate in each computation step, reducing both communication overhead and energy consumption. In contrast, dense connectivity patterns require large numbers of synaptic interactions to be evaluated at each update, effectively removing the sparsity that enables efficient event-driven execution. As a result, dense linear transformations behave similarly to conventional dense neural network layers, limiting the potential advantages of neuromorphic hardware for such workloads.

Furthermore, collapsing the entire ADMM iteration into a single linear operator increases the dimensionality of the transformation. For a problem with N decision variables and N_c constraints, the resulting transformation involves a matrix of dimension approximately $N + 2N_c$. These considerations motivate the exploration of alternative mappings that preserve the structured sparsity of the underlying optimization problem.

6.4 Mapping Structured Optimization Algorithms

Instead of collapsing the entire optimization iteration into a single dense transformation, an alternative approach is to map the individual computational steps of the optimization algorithm directly onto the neuromorphic architecture.

The FPGA implementation described in previous chapters follows this structured approach. In that implementation, the ADMM algorithm is decomposed into a sequence of simpler computational blocks, including matrix–vector multiplications, projections, and a structured linear system solve.

Among these operations, the dominant computational cost arises from solving the linear system associated with the KKT conditions. If the Cholesky factorization of the system matrix is computed offline, the runtime computation reduces to forward and backward substitution.

Forward substitution computes the vector y as

$$y_i = \frac{1}{L_{ii}} \left(b_i - \sum_{j=1}^{i-1} L_{ij} y_j \right), \tag{6.2}$$

where L is the lower triangular Cholesky factor. This computation has a structured dependency graph: each variable y_i depends only on previously computed variables.

This dependency structure can be mapped to a directed neural network in which each neuron represents a variable and receives inputs corresponding to the nonzero elements of the triangular matrix. Figure 6.4 illustrates this mapping for the case of a matrix with bandwidth 2, where no connection exists between y_1 and y_4.

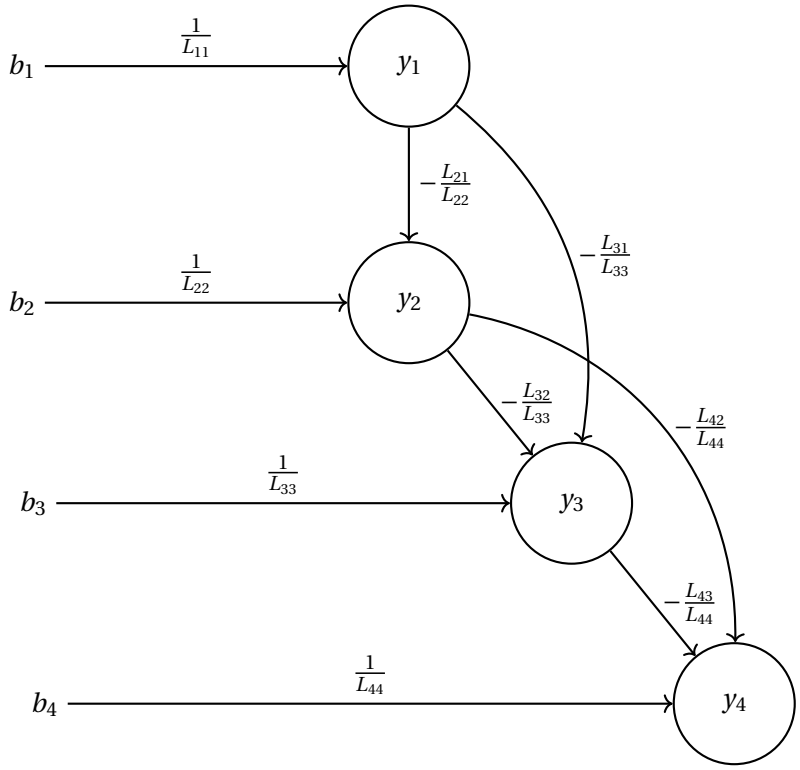


Figure 6.4: Conceptual mapping of forward substitution onto a spiking neural network.

Because the triangular matrix remains sparse, the corresponding neural network connectivity is also sparse. This sparsity may allow the neuromorphic architecture to exploit its event-driven execution model more effectively than dense formulations.

6.5 Quantization and Weight Representation

A major challenge in implementing numerical algorithms on Loihi 2 arises from the quantized representation of synaptic weights. While many machine learning models tolerate reduced numerical precision [81, 82], iterative optimization algorithms may accumulate quantization errors across multiple iterations.

To represent a matrix A using the Loihi weight format, each entry must be approximated as

$$\widehat{A}_{ij} = m_{ij} 2^{e_{ij}},$$

where m_{ij} is an integer mantissa and e_{ij} is selected from a limited set of exponent groups. Selecting

these parameters while minimizing approximation error can be formulated as

$$\begin{aligned} \min_{\{m_{ij}\},\mathscr{E}} \quad & \|A-\widehat{A}\|_2 \\ \text{s.t.} \quad & \widehat{A}_{ij} = m_{ij}2^{e_{ij}}, \\ & m_{ij} \in \{-2^7,\dots,2^7-1\}, \\ & e_{ij} \in \mathscr{E}, \\ & |\mathscr{E}| = 16. \end{aligned}$$

This problem is a mixed-integer nonlinear optimization problem. Solving it exactly becomes computationally expensive for large matrices. In practice, the matrix can therefore be partitioned into smaller blocks and the optimization applied independently within each block.

This binning strategy reduces computational complexity while capturing much of the dynamic range of the matrix. Once exponent groups are selected, mantissas can be computed to minimize the approximation error within each block.

6.6 Preliminary Experiments

Access to Loihi 2 hardware was limited during the development of this work. Nevertheless, the numerical effects of the weight quantization can be approximated in simulation.

In these experiments, the matrices used by the optimization algorithm were first mapped to the Loihi representation using the optimization procedure described in the previous section. The resulting weights were stored as floating-point values. During simulation, intermediate computations were periodically quantized in the fixed-point spike format and then converted back to floating-point values. This procedure approximates the numerical effects introduced by the hardware representation.

The simulations were executed on a GPU in order to exploit parallel computation while emulating the quantization constraints. Although this approach does not reproduce the exact behavior of the hardware, it allows an initial evaluation of the algorithm's numerical robustness.

The results indicate that the optimization algorithm retains its convergence behavior provided that the exponent binning strategy achieves sufficient approximation accuracy. When the number of bins is too small, quantization errors can accumulate and degrade convergence.

Figure 6.5 shows an example closed-loop response of the quadrotor controller under a step input when the quantized representation is used.

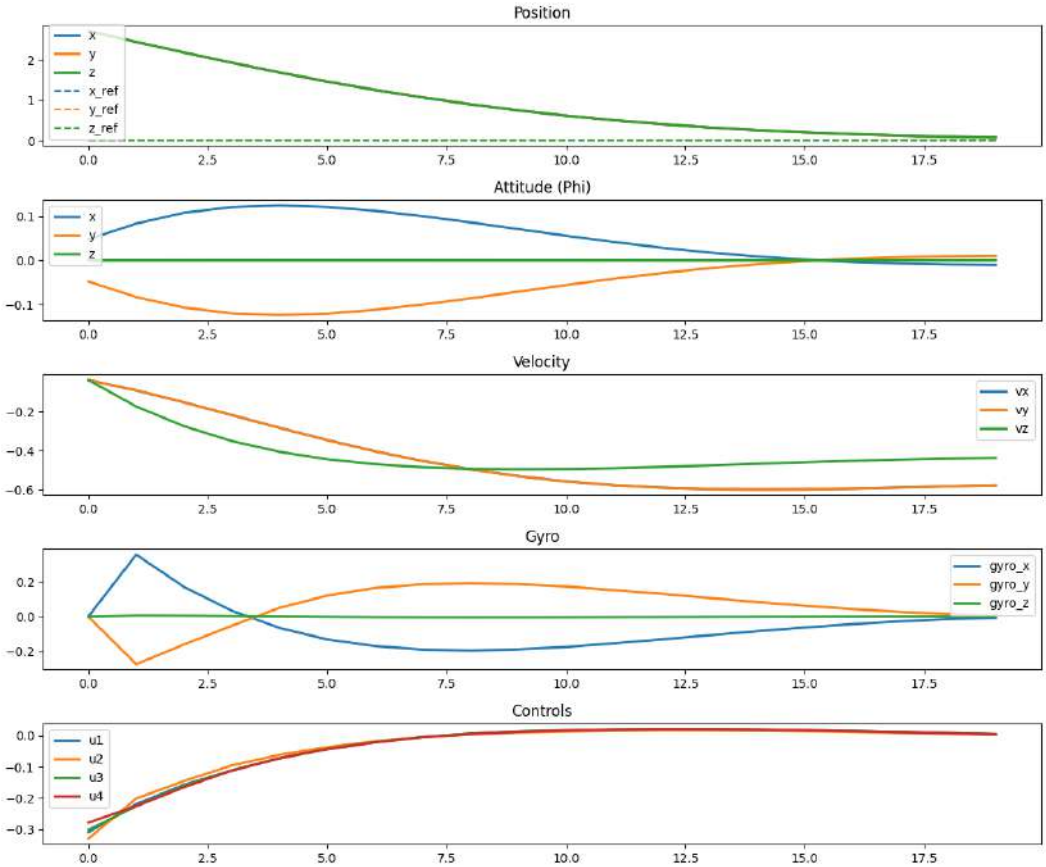


Figure 6.5: Closed-loop response of the controller using quantized weight representations.

6.7 Discussion

This chapter explored the feasibility of implementing optimization algorithms for model predictive control on neuromorphic hardware. Rather than presenting a complete implementation, the goal was to identify architectural opportunities and algorithmic challenges that arise when mapping structured optimization algorithms to neuromorphic processors such as Loihi 2.

The analysis highlights several observations. First, existing neuromorphic formulations of quadratic programming, such as ReLU-QP, express entire optimization iterations as dense matrix–vector transformations. While this formulation is convenient from an implementation perspective, it does not preserve the structured sparsity that characterizes the quadratic programs arising in MPC. In particular, constructing a single operator for the ADMM iteration requires embedding the inverse of the KKT matrix, which typically results in dense connectivity even when the original problem structure is sparse.

Second, preserving the algorithmic structure of the optimization method may be more compatible with neuromorphic architectures. By decomposing the algorithm into its constituent operations, the structured linear system solve remains the dominant computational component. Mapping triangular solves arising from a Cholesky factorization to neural connectivity patterns preserves sparsity and yields a dependency graph that can be represented by local neuron interactions.

Third, the numerical representation used by Loihi introduces additional constraints that must be considered when implementing iterative optimization algorithms. The mantissa–exponent weight format requires careful selection of scaling factors to minimize approximation error. The proposed optimization-based weight mapping provides a practical approach to representing system matrices within these constraints.

Preliminary experiments using GPU-based simulations of the quantized weight representation indicate that the convergence behavior of the optimization algorithm can be preserved when the weight approximation achieves sufficient accuracy. Although these simulations do not reproduce the exact execution model of the hardware, they provide initial evidence that iterative optimization algorithms may remain numerically stable under the quantization constraints imposed by neuromorphic processors.

Overall, these results suggest that neuromorphic architectures may provide a viable platform for structured optimization algorithms, particularly when the algorithmic formulation preserves sparsity and local computational dependencies. Further investigation on actual neuromorphic hardware is required to evaluate performance, latency, and energy efficiency in realistic control applications.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

This thesis investigated how hardware–algorithm co-design can improve the practicality of model predictive control on resource-constrained robotic platforms. The central challenge is that linear MPC requires the repeated solution of a quadratic program within the tight timing constraints of a high-rate control loop, while small aerial robots provide only limited onboard computational resources. This limitation is particularly pronounced on platforms such as the Crazyflie nano-quadrotor, where the optimization must be executed under strict constraints on latency, memory, power, and weight.

To address this problem, this work proposed an ADMM-based linear MPC architecture in which the computationally intensive optimization steps are offloaded to an FPGA, while the remaining control pipeline is maintained on the microcontroller. The approach combined structure-exploiting solver design, fixed-point arithmetic, compressed matrix storage, and hardware-oriented implementation techniques to obtain a low-latency and predictable execution architecture suitable for embedded deployment. In addition to the solver itself, the work included the design of a custom FPGA expansion board, the integration of the accelerator within the Crazyflie firmware stack, and the deployment of the complete system on a flying platform.

The experimental results show that this approach substantially expands the practically achievable operating region of onboard linear MPC on the Crazyflie. Compared with the MCU-only TinyMPC baseline [61], the FPGA-accelerated implementation enables significantly lower solve times, which in turn allows the use of longer prediction horizons and larger iteration budgets within the same control period. This advantage is particularly important in constrained control scenarios, where effective optimization requires sufficient progress of the iterative solver at every update. The flight experiments demonstrated that the proposed system can execute constrained MPC fully onboard at

a control rate of 500,Hz, while preserving stable closed-loop operation on a resource-constrained nano-quadrotor.

Beyond the specific quadrotor application, the main contribution of this thesis is to show that the limitations of embedded MPC are not solely algorithmic, but also architectural. Efficient optimization on small robotic platforms depends not only on the choice of solver, but also on how the computational structure of the problem is matched to the underlying hardware. In this context, FPGA acceleration provides an effective middle ground between general-purpose embedded processors and larger high-performance computing platforms, enabling low-latency optimization while remaining compatible with edge deployment.

In addition to the FPGA-based implementation, this thesis also presented a preliminary exploratory study on neuromorphic hardware as an alternative computing substrate for optimization-based control. Although this direction remains at an early stage, the study suggests that emerging architectures such as Intel Loihi 2 may offer interesting opportunities for future low-power implementations of iterative optimization algorithms.

Overall, this work demonstrates that hardware–algorithm co-design is a promising approach for bringing optimization-based control closer to practical deployment on severely resource-constrained robots. By combining a structured MPC formulation with a dedicated hardware accelerator and validating the resulting system in real flight experiments, this thesis provides a concrete step toward fully embedded real-time optimization for robotics.

7.2 Future Work

While the proposed system demonstrates that hardware–algorithm co-design can enable real-time linear MPC on highly resource-constrained robotic platforms, several opportunities remain for further improvement and extension of the approach. These opportunities can be broadly divided into engineering improvements to the current system architecture and longer-term research directions.

7.2.1 Engineering Improvements

Several practical improvements could further enhance the efficiency and flexibility of the current implementation.

First, the communication interface between the microcontroller and the FPGA could be improved. The current system relies on a half-duplex SPI communication scheme, which requires alternating transmission phases between the two devices. Implementing full-duplex communication would allow state updates and control outputs to be exchanged simultaneously, reducing communication

latency and simplifying the interaction between the solver and the flight control stack.

Second, trajectory handling could be improved to reduce the reliance on on-chip FPGA memory. In the current implementation, reference trajectories can be stored directly in the FPGA in order to minimize communication overhead. While this approach is effective, it consumes valuable BRAM resources and limits the maximum trajectory length. A more flexible approach would stream trajectory information directly from the microcontroller, which may itself receive trajectory updates from external sources or onboard planning modules. With efficient use of the SPI interface, the available communication bandwidth is sufficient to transmit trajectory information without becoming the primary bottleneck.

Finally, additional improvements in the internal memory representation of the solver could reduce resource usage. In the current implementation, several matrices are stored explicitly in both original and transposed form in order to simplify access patterns during solver iterations. Future implementations could reduce memory usage by avoiding redundant storage and exploiting the structured nature of the MPC problem. In particular, solvers based on Riccati recursions or other multistage optimization techniques could leverage the horizon structure of the optimal control problem to avoid storing large block matrices explicitly and instead reuse the system matrices across stages.

7.2.2 Future Research Directions

Beyond incremental engineering improvements, several research directions could further extend the capabilities of the proposed hardware–algorithm co-design approach.

Asynchronous MPC Execution and Energy-Aware Scheduling

The current controller operates at a fixed control rate and executes a predetermined number of ADMM iterations at each control step. While this approach provides predictable timing behavior, it does not account for the fact that the difficulty of the optimization problem can vary significantly across time steps. When the system operates near equilibrium, the warm-started solution may already be close to optimal, whereas constraint activation or large disturbances may require additional iterations to reach convergence.

Future work could explore asynchronous MPC execution schemes in which the solver runs continuously and the controller applies the most recent available solution. When a new solution becomes available it is immediately deployed, while in situations where convergence requires additional time the controller continues applying the previously computed control input. Such an approach would allow the computational effort to adapt dynamically to the complexity of the optimization problem.

This framework could also enable improved energy efficiency. When the warm-started solution

converges quickly, the solver could terminate early and remain idle until new information becomes available, avoiding unnecessary computation. Additionally, the operating frequency of the accelerator could be adjusted dynamically depending on the required computational effort. If fewer solver iterations are needed, the clock frequency could be reduced while still completing the computation within the control period. Since dynamic power consumption scales approximately linearly with clock frequency, such adaptive frequency scaling could significantly reduce power consumption while maintaining the same control update rate.

Neuromorphic and Optimization-Based Control Integration

Another promising direction is the integration of optimization-based control onto neuromorphic computing platforms.

This direction could enable the deployment of the MPC controller directly within a neuromorphic substrate, potentially allowing optimization-based control to be executed with extremely low power consumption. The optimization algorithm itself could be mapped onto neuromorphic primitives, enabling an end-to-end control pipeline implemented entirely on neuromorphic hardware. Such an approach could significantly reduce the energy cost of real-time control on resource-constrained robots.

In addition, neuromorphic systems naturally interface with event-based sensors, which could enable low-latency and energy-efficient perception pipelines integrated with the MPC controller, while preserving the constraint handling and safety guarantees provided by optimization-based control.

Structure-Exploiting MPC Solvers

The solver presented in this work relies on a generic linear system formulation of the ADMM iterations. While this approach simplifies the hardware architecture, it does not fully exploit the multistage structure of the optimal control problem.

Future work could investigate solvers that explicitly exploit the horizon structure of MPC problems. Methods based on Riccati recursions or similar forward–backward factorizations allow the linear system to be solved through structured operations along the prediction horizon. Such approaches can significantly reduce memory requirements by reusing system matrices across stages and avoiding redundant storage of transposed matrices.

Adopting structure-exploiting solvers could therefore improve both the memory efficiency and scalability of the hardware accelerator.

Nonlinear MPC on Resource-Constrained Platforms

While the proposed system demonstrates very high performance for linear MPC, the quadrotor application considered in this work does not fully exploit the computational capabilities of the accelerator. In practice, the main limitation of quadrotor control on platforms such as the Crazyflie is not the ability to solve linear MPC problems faster or with longer horizons, but rather the accuracy of the linear model used to represent the system dynamics.

Quadrotor dynamics are inherently nonlinear, particularly during aggressive maneuvers or when operating far from the linearization point. As a result, improving the representation of the system dynamics is likely to provide greater benefits than further increasing the performance of linear MPC on this platform.

A promising direction is therefore the development of hardware–algorithm co-designed implementations of nonlinear MPC. In such a framework, the accelerator could solve the quadratic subproblems arising in nonlinear MPC methods such as sequential quadratic programming or real-time iteration schemes, while the microcontroller performs model linearization and trajectory updates. This approach would allow the controller to better capture the nonlinear dynamics of the system while maintaining real-time execution on resource-constrained robotic platforms.

Scaling to Higher-Dimensional Systems

The quadrotor platform used in this work results in a relatively small MPC problem. While this makes it a suitable benchmark for demonstrating real-time onboard control, the advantages of hardware acceleration may become even more significant for larger systems.

Many robotic applications rely on linear MPC models with significantly higher state dimensionality, including legged locomotion and humanoid balance control. Despite the underlying nonlinear dynamics, linear MPC formulations are often employed due to their computational tractability and favorable control properties, including legged locomotion and humanoid balance control [83, 84].

Applying the proposed hardware–algorithm co-design approach to such systems could therefore enable real-time MPC execution for higher-dimensional problems on embedded platforms and further demonstrate the scalability of the architecture.

Application-Specific Integrated Circuit Implementations

While FPGAs provide a flexible platform for rapid prototyping and hardware–algorithm co-design, the architecture presented in this work could also be translated to an application-specific integrated circuit (ASIC). An ASIC implementation would allow the solver architecture to be optimized more

aggressively by removing the overhead associated with the programmable fabric of an FPGA.

Dedicated arithmetic units, optimized memory hierarchies, and custom data paths could be designed specifically for the structure of the MPC solver, enabling higher computational efficiency and lower power consumption. Such improvements would be particularly valuable for small autonomous robots, where energy efficiency is a critical constraint.

In the longer term, ASIC implementations could also enable tighter integration with sensing and processing components on the same chip, potentially leading to fully integrated control accelerators capable of executing optimization-based control algorithms within highly constrained embedded systems.

Bibliography

[1] J. Rawlings, D.Q. Mayne, and Moritz Diehl. *Model Predictive Control: Theory, Computation, and Design*. Jan. 2017.

[2] E. F. Camacho and C. Bordons. *Model Predictive Control*. Ed. by Michael J. Grimble and Michael A. Johnson. Advanced Textbooks in Control and Signal Processing. London: Springer, 2007. ISBN: 978-1-85233-694-3. DOI: 10.1007/978-0-85729-398-5. (Visited on 03/10/2026).

[3] D. Q. Mayne, J. B. Rawlings, C. V. Rao, and P. O. M. Scokaert. "Constrained Model Predictive Control: Stability and Optimality". In: *Automatica* 36.6 (June 2000), pp. 789–814. ISSN: 0005-1098. DOI: 10.1016/S0005-1098(99)00214-9. (Visited on 03/10/2026).

[4] S. Joe Qin and Thomas A. Badgwell. "A Survey of Industrial Model Predictive Control Technology". In: *Control Engineering Practice* 11.7 (July 2003), pp. 733–764. ISSN: 0967-0661. DOI: 10.1016/S0967-0661(02)00186-7. (Visited on 03/10/2026).

[5] Stephen Boyd and Lieven Vandenberghe. *Convex Optimization*. Cambridge university press, 2004. (Visited on 03/12/2026).

[6] Jorge Nocedal and Stephen Wright. *Numerical Optimization*. Springer Science & Business Media, Dec. 2006. ISBN: 978-0-387-40065-5.

[7] Wojciech Giernacki, Mateusz Skwierczyński, Wojciech Witwicki, Paweł Wroński, and Piotr Kozierski. "Crazyflie 2.0 Quadrotor as a Platform for Research and Education in Robotics and Control Engineering". In: *2017 22nd International Conference on Methods and Models in Automation and Robotics (MMAR)*. IEEE, 2017, pp. 37–42.

[8] Vivek Adajania, Siqi Zhou, Singh Arun, and Angela Schoellig. "AMSwarm: An Alternating Minimization Approach for Safe Motion Planning of Quadrotor Swarms in Cluttered Environments". In: *2023 IEEE International Conference on Robotics and Automation (ICRA)*. 2023, pp. 1421–1427.

[9] Pratyush Varshney, Gajendra Nagar, and Indranil Saha. "DeepControl: Energy-efficient Control of a Quadrotor Using a Deep Neural Network". In: *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2019, pp. 43–50. DOI: 10.1109/IROS40897.2019.8968236.

[10] Nathan O Lambert, Daniel S Drew, Joseph Yaconelli, Sergey Levine, Roberto Calandra, and Kristofer SJ Pister. "Low-Level Control of a Quadrotor with Deep Model-Based Reinforcement Learning". In: *IEEE Robotics and Automation Letters* 4.4 (2019), pp. 4224–4230.

[11] Carlos E Luis, Marijan Vukosavljev, and Angela P Schoellig. "Online Trajectory Generation with Distributed Model Predictive Control for Multi-Robot Motion Planning". In: *IEEE Robotics and Automation Letters* 5.2 (2020), pp. 604–611.

[12] Lele Xi, Xinyi Wang, Lei Jiao, Shupeng Lai, Zhihong Peng, and Ben M Chen. "GTO-MPC-based Target Chasing Using a Quadrotor in Cluttered Environments". In: *IEEE Transactions on Industrial Electronics* 69.6 (2021), pp. 6026–6035.

[13] Guillem Torrente, Elia Kaufmann, Philipp Föhn, and Davide Scaramuzza. "Data-Driven MPC for Quadrotors". In: *IEEE Robotics and Automation Letters* 6.2 (2021), pp. 3769–3776.

[14] Kong Yao Chee, Tom Z Jiahao, and M Ani Hsieh. "Knode-Mpc: A Knowledge-Based Data-Driven Predictive Control Framework for Aerial Robots". In: *IEEE Robotics and Automation Letters* 7.2 (2022), pp. 2819–2826.

[15] Brian Plancher and Scott Kuindersma. "A Performance Analysis of Parallel Differential Dynamic Programming on a Gpu". In: *Algorithmic Foundations of Robotics XIII: Proceedings of the 13th Workshop on the Algorithmic Foundations of Robotics 13*. Springer, 2020, pp. 656–672.

[16] Sabrina M. Neuman, Brian Plancher, Thomas Bourgeat, Thierry Tambe, Srinivas Devadas, and Vijay Janapa Reddi. "Robomorphic Computing: A Design Methodology for Domain-Specific Accelerators Parameterized by Robot Morphology". In: Asplos 2021. New York, NY, USA: Association for Computing Machinery, 2021, pp. 674–686. ISBN: 978-1-4503-8317-2. DOI: 10.1145/3445814.3446746.

[17] Khai Nguyen, Sam Schoedel, Anoushka Alavilli, Brian Plancher, and Zachary Manchester. "TinyMPC: Model-Predictive Control on Resource-Constrained Microcontrollers". In: *IEEE International Conference on Robotics and Automation, ICRA 2024, Yokohama, Japan, May 13-17, 2024*. IEEE, 2024, pp. 1–7. DOI: 10.1109/ICRA57147.2024.10610987.

[18] Roland Schwan, Daniel Kuhn, and Colin N. Jones. *Exploiting Multistage Optimization Structure in Proximal Solvers*. Sept. 2025. DOI: 10.48550/arXiv.2503.12664. (Visited on 03/11/2026).

[19] Bartolomeo Stellato, Goran Banjac, Paul Goulart, Alberto Bemporad, and Stephen Boyd. "OSQP: An Operator Splitting Solver for Quadratic Programs". In: *Mathematical Programming Computation* 12.4 (Dec. 2020), pp. 637–672. ISSN: 1867-2957. DOI: 10.1007/s12532-020-00179-2. (Visited on 02/09/2026).

[20] Stephen Boyd, Neal Parikh, Eric Chu, Borja Peleato, and Jonathan Eckstein. "Distributed Optimization and Statistical Learning via the Alternating Direction Method of Multipliers". In: *Foundations and Trends in Machine Learning* 3 (Jan. 2011), pp. 1–122. DOI: 10.1561/2200000016.

[21] H. J. Ferreau, S. Almér, R. Verschueren, M. Diehl, D. Frick, A. Domahidi, J. L. Jerez, G. Stathopoulos, and C. Jones. "Embedded Optimization Methods for Industrial Automatic Control*". In: *IFAC-PapersOnLine*. 20th IFAC World Congress 50.1 (July 2017), pp. 13194–13209. ISSN: 2405-8963. DOI: 10.1016/j.ifacol.2017.08.1946. (Visited on 03/10/2026).

[22] Scott Hauck and André DeHon. *Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation*. Elsevier, July 2010. ISBN: 978-0-08-055601-7.

[23] Ian Kuon, Russell Tessier, and Jonathan Rose. "FPGA Architecture: Survey and Challenges". In: *Foundations and Trends® in Electronic Design Automation* 2.2 (2008), pp. 135–253. ISSN: 1551-3939, 1551-3947. DOI: 10.1561/1000000005. (Visited on 03/10/2026).

[24] Samo Gerkšič and Boštjan Pregelj. "Finite-Word-Length FPGA Implementation of Model Predictive Control for ITER Resistive Wall Mode Control". In: *Fusion Engineering and Design* 169 (Aug. 2021), p. 112480. ISSN: 0920-3796. DOI: 10.1016/j.fusengdes.2021.112480. (Visited on 02/09/2026).

[25] Edward Nicholas Hartley, Juan Luis Jerez, Andrea Suardi, Jan M. Maciejowski, Eric C. Kerrigan, and George A. Constantinides. "Predictive Control Using an FPGA With Application to Aircraft Control". In: *IEEE Transactions on Control Systems Technology* 22.3 (May 2014), pp. 1006–1017. ISSN: 1558-0865. DOI: 10.1109/TCST.2013.2271791. (Visited on 02/09/2026).

[26] Juan L. Jerez, Paul J. Goulart, Stefan Richter, George A. Constantinides, Eric C. Kerrigan, and Manfred Morari. "Embedded Online Optimization for Model Predictive Control at Megahertz Rates". In: *IEEE Transactions on Automatic Control* 59.12 (Dec. 2014), pp. 3238–3251. ISSN: 1558-2523. DOI: 10.1109/TAC.2014.2351991. (Visited on 02/09/2026).

[27] Juan L. Jerez, George A. Constantinides, and Eric C. Kerrigan. "FPGA Implementation of an Interior Point Solver for Linear Model Predictive Control". In: *2010 International Conference on Field-Programmable Technology*. Dec. 2010, pp. 316–319. DOI: 10.1109/FPT.2010.5681439. (Visited on 02/09/2026).

[28] Juan Luis Jerez, George Anthony Constantinides, and Eric C. Kerrigan. "An FPGA Implementation of a Sparse Quadratic Programming Solver for Constrained Predictive Control". In: *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. FPGA '11. New York, NY, USA: Association for Computing Machinery, Feb. 2011, pp. 209–218. ISBN: 978-1-4503-0554-9. DOI: 10.1145/1950413.1950454. (Visited on 02/09/2026).

[29] Min Jeong, Manuel Schoen, and Jürgen Biela. "When FPGAs Meet ADMM with High-level Synthesis (HLS): A Real-time Implementation of Long-horizon MPC for Power Electronic Systems". In: *2023 11th International Conference on Power Electronics and ECCE Asia (ICPE 2023 - ECCE Asia)*. May 2023, pp. 1704–1711. DOI: 10.23919/ICPE2023-ECCEAsia54778.2023.10213796. (Visited on 02/09/2026).

[30] Junyi Liu, Helfried Peyrl, Andreas Burg, and George A. Constantinides. “FPGA Implementation of an Interior Point Method for High-Speed Model Predictive Control”. In: *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*. Sept. 2014, pp. 1–8. DOI: 10.1109/FPL.2014.6927473. (Visited on 02/09/2026).

[31] Harsh A. Shukla, Bulat Khusainov, Eric C. Kerrigan, and Colin N. Jones. “Software and Hardware Code Generation for Predictive Control Using Splitting Methods*”. In: *IFAC-PapersOnLine*. 20th IFAC World Congress 50.1 (July 2017), pp. 14386–14391. ISSN: 2405-8963. DOI: 10.1016/j.ifacol.2017.08.2025. (Visited on 02/09/2026).

[32] Saket Adhau, Kiran Phalke, Apeksha Nalawade, Deepak Ingole, Sayli Patil, and Dayaram Sonawane. “Implementation and Analysis of Offset-Free Explicit Model Predictive Controller on FPGA”. In: *2019 Fifth Indian Control Conference (ICC)*. Jan. 2019, pp. 231–236. DOI: 10.1109/INDIANCC.2019.8715619. (Visited on 02/09/2026).

[33] Deepak Ingole, Juraj Holaza, Bálint Takács, and Michal Kvasnica. “FPGA-based Explicit Model Predictive Control for Closed-Loop Control of Intravenous Anesthesia”. In: *2015 20th International Conference on Process Control (PC)*. June 2015, pp. 42–47. DOI: 10.1109/PC.2015.7169936. (Visited on 02/09/2026).

[34] Min Jeong, Simon Fuchs, and Jürgen Biela. “When FPGAs Meet Regionless Explicit MPC: An Implementation of Long-horizon Linear MPC for Power Electronic Systems”. In: *IECON 2020 The 46th Annual Conference of the IEEE Industrial Electronics Society*. Oct. 2020, pp. 3085–3092. DOI: 10.1109/IECON43393.2020.9254277. (Visited on 02/09/2026).

[35] P. Tondel, T.A. Johansen, and A. Bemporad. “Computation and Approximation of Piecewise Affine Control Laws via Binary Search Trees”. In: *Proceedings of the 41st IEEE Conference on Decision and Control, 2002*. Vol. 3. Dec. 2002, 3144–3149 vol.3. DOI: 10.1109/CDC.2002.1184353. (Visited on 02/09/2026).

[36] Zhenyu Wu, Brian Plancher, Ian McInerney, Hayden Kwok-Hay So, Maolin Wang, and Kwang-Ting Cheng. “MPC Solver Hardware Generation Framework with Model-Specific Operation Fusion and Pruning”. In: *2025 International Conference on Field Programmable Technology (ICFPT)*. Dec. 2025, pp. 239–247. DOI: 10.1109/ICFPT67023.2025.00047. (Visited on 02/16/2026).

[37] Geoff Knagge, Adrian Wills, Adam Mills, and Brett Ninness. “ASIC and FPGA Implementation Strategies for Model Predictive Control”. In: *2009 European Control Conference (ECC)*. Aug. 2009, pp. 144–149. DOI: 10.23919/ECC.2009.7074394. (Visited on 02/09/2026).

[38] James B Aimone. “Neuromorphic Computing: A Theoretical Framework for Time, Space, and Energy Scaling”. In: (2025). DOI: 10.48550/ARXIV.2507.17886. (Visited on 02/08/2026).

[39] Mike Davies, Narayan Srinivasa, Tsung-Han Lin, Gautham Chinya, Yongqiang Cao, Sri Harsha Choday, Georgios Dimou, Prasad Joshi, Nabil Imam, Shweta Jain, Yuyun Liao, Chit-Kwan Lin, Andrew Lines, Ruokun Liu, Deepak Mathaikutty, Steven McCoy, Arnab Paul, Jonathan Tse, Guruguhanathan Venkataramanan, Yi-Hsin Weng, Andreas Wild, Yoonseok Yang, and Hong

Wang. “Loihi: A Neuromorphic Manycore Processor with On-Chip Learning”. In: *IEEE Micro* 38.1 (Jan. 2018), pp. 82–99. ISSN: 1937-4143. DOI: 10.1109/MM.2018.112130359. (Visited on 03/10/2026).

[40] H.J. Ferreau, C. Kirches, A. Potschka, H.G. Bock, and M. Diehl. “qpOASES: A Parametric Active-Set Algorithm for Quadratic Programming”. In: *Mathematical Programming Computation* 6.4 (2014), pp. 327–363.

[41] Roland Schwan, Yuning Jiang, Daniel Kuhn, and Colin N. Jones. “PIQP: A Proximal Interior-Point Quadratic Programming Solver”. In: *2023 62nd IEEE Conference on Decision and Control (CDC)*. Dec. 2023, pp. 1088–1093. DOI: 10.1109/CDC49753.2023.10383915. (Visited on 03/12/2026).

[42] Dimitris Kouzoupis, Hans Joachim Ferreau, Helfried Peyrl, and Moritz Diehl. “First-Order Methods in Embedded Nonlinear Model Predictive Control”. In: *14th European Control Conference, ECC 2015, Linz, Austria, July 15-17, 2015*. IEEE, 2015, pp. 2617–2622. DOI: 10.1109/ECC.2015.7330932.

[43] Mark D. Hill and Michael R. Marty. “Amdahl’s Law in the Multicore Era”. In: *Computer* 41.7 (July 2008), pp. 33–38. ISSN: 1558-0814. DOI: 10.1109/MC.2008.209. (Visited on 03/12/2026).

[44] *Computer Architecture*. Nov. 2017. ISBN: 978-0-12-811905-1. (Visited on 03/10/2026).

[45] Peter Marwedel. *Embedded System Design: Embedded Systems Foundations of Cyber-Physical Systems, and the Internet of Things*. Embedded Systems. Cham: Springer International Publishing, 2021. ISBN: 978-3-030-60909-2. DOI: 10.1007/978-3-030-60910-8. (Visited on 03/10/2026).

[46] Michael Barr Massa Anthony. *Programming Embedded Systems, 2nd Edition*. ISBN: 978-0-596-00983-0. (Visited on 03/10/2026).

[47] Sam Schoedel, Khai Nguyen, Elakhya Nedumaran, Brian Plancher, and Zachary Manchester. “Code Generation for Conic Model-Predictive Control on Microcontrollers with TinyMPC”. In: *CoRR* abs/2403.18149 (2024). DOI: 10.48550/ARXIV.2403.18149. arXiv: 2403.18149. (Visited on 02/09/2026).

[48] *Programming Massively Parallel Processors*. Aug. 2022. ISBN: 978-0-323-91231-0. (Visited on 03/10/2026).

[49] John Owens. “GPU Architecture Overview”. In: *ACM SIGGRAPH 2007 Courses*. SIGGRAPH ’07. New York, NY, USA: Association for Computing Machinery, Aug. 2007, 2–es. ISBN: 978-1-4503-1823-5. DOI: 10.1145/1281500.1281643. (Visited on 03/10/2026).

[50] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. “Scalable Parallel Programming with CUDA”. In: *ACM SIGGRAPH 2008 Classes*. SIGGRAPH ’08. New York, NY, USA: Association for Computing Machinery, Aug. 2008, pp. 1–14. ISBN: 978-1-4503-7845-1. DOI: 10.1145/1401132.1401152. (Visited on 03/10/2026).

[51] Michel Schubiger, Goran Banjac, and John Lygeros. “GPU Acceleration of ADMM for Large-Scale Quadratic Programming”. In: *Journal of Parallel and Distributed Computing* 144 (Oct. 2020), pp. 55–67. ISSN: 0743-7315. DOI: 10.1016/j.jpdc.2020.05.021. (Visited on 02/08/2026).

[52] Emre Adabag, Miloni Atal, William Gerard, and Brian Plancher. “MPCGPU: Real-Time Nonlinear Model Predictive Control through Preconditioned Conjugate Gradient on the GPU”. In: *IEEE International Conference on Robotics and Automation, ICRA 2024, Yokohama, Japan, May 13-17, 2024*. IEEE, 2024, pp. 9787–9794. DOI: 10.1109/ICRA57147.2024.10611212.

[53] Alexander Du, Emre Adabag, Gabriel Bravo, and Brian Plancher. *GATO: GPU-Accelerated and Batched Trajectory Optimization for Scalable Edge Model Predictive Control*. https://arxiv.org/abs/2510.07625v1. Oct. 2025. (Visited on 02/08/2026).

[54] *Digital Design: With an Introduction to the Verilog HDL, VHDL, and SystemVerilog (Pearson+) | Florida Gulf Coast University Bookstore*. https://fgcu.bncollege.com/c/Digital-Design-With-an-Introduction-to-the-Verilog-HDL-VHDL-and-SystemVerilog-Pearson/p/MBS_11216402_dg. (Visited on 03/10/2026).

[55] “7 Series FPGAs Data Sheet: Overview (DS180)”. In: (2020).

[56] Sharif Azem, David Scheunert, Mengguang Li, Jonas Gehrunger, Kai Cui, Christian Hochberger, and Heinz Koeppl. *FPGA-Based Neural Thrust Controller for UAVs*. https://arxiv.org/abs/2403.18703v2. Mar. 2024. (Visited on 02/08/2026).

[57] Ashish Rao Mangalore, Gabriel Andres Fonseca Guerra, Sumedh R. Risbud, Philipp Stratmann, and Andreas Wild. “Neuromorphic Quadratic Programming for Efficient and Scalable Model Predictive Control: Towards Advancing Speed and Energy Efficiency in Robotic Control”. In: *IEEE Robotics & Automation Magazine* 32.2 (June 2025), pp. 69–79. ISSN: 1558-223X. DOI: 10.1109/MRA.2024.3415005. (Visited on 03/02/2026).

[58] Samir Bouabdallah and Roland Siegwart. “Full Control of a Quadrotor”. In: *2007 IEEE/RSJ International Conference on Intelligent Robots and Systems*. Oct. 2007, pp. 153–158. DOI: 10.1109/IROS.2007.4399042. (Visited on 03/11/2026).

[59] Kimberly Mcguire, Christophe De Wagter, Karl Tuyls, H. Kappen, and Guido Croon. “Minimal Navigation Solution for a Swarm of Tiny Flying Robots to Explore an Unknown Environment”. In: *Science Robotics* 4 (Oct. 2019), eaaw9710. DOI: 10.1126/scirobotics.aaw9710.

[60] Jemin Hwangbo, Inkyu Sa, Roland Siegwart, and Marco Hutter. “Control of a Quadrotor With Reinforcement Learning”. In: *IEEE Robotics and Automation Letters* 2.4 (Oct. 2017), pp. 2096–2103. ISSN: 2377-3766. DOI: 10.1109/LRA.2017.2720851. (Visited on 03/11/2026).

[61] Huan Nguyen, Mina Kamel, Kostas Alexis, and Roland Siegwart. “Model Predictive Control for Micro Aerial Vehicles: A Survey”. In: *2021 European Control Conference (ECC)*. June 2021, pp. 1556–1563. DOI: 10.23919/ECC54610.2021.9654841. (Visited on 02/16/2026).

[62] Guanrui Li, Alex Tunchez, and Giuseppe Loianno. "PCMPC: Perception-Constrained Model Predictive Control for Quadrotors with Suspended Loads Using a Single Camera and IMU". In: *2021 IEEE International Conference on Robotics and Automation (ICRA)*. May 2021, pp. 2012–2018. DOI: 10.1109/ICRA48506.2021.9561449. (Visited on 03/11/2026).

[63] Alexandre Didier, Anilkumar Parsi, Jeremy Coulson, and Roy S. Smith. "Robust Adaptive Model Predictive Control of Quadrotors". In: *2021 European Control Conference (ECC)*. June 2021, pp. 657–662. DOI: 10.23919/ECC54610.2021.9654893. (Visited on 03/11/2026).

[64] Bárbara Barros Carlos, Tommaso Sartor, Andrea Zanelli, Gianluca Frison, Wolfram Burgard, Moritz Diehl, and Giuseppe Oriolo. "An Efficient Real-Time NMPC for Quadrotor Position Control under Communication Time-Delay". In: *2020 16th International Conference on Control, Automation, Robotics and Vision (ICARCV)*. Dec. 2020, pp. 982–989. DOI: 10.1109/ICARCV50220.2020.9305513. (Visited on 03/11/2026).

[65] Angel Romero, Sihao Sun, Philipp Foehn, and Davide Scaramuzza. "Model Predictive Contouring Control for Time-Optimal Quadrotor Flight". In: *IEEE Transactions on Robotics* 38.6 (2022), pp. 3340–3356.

[66] Brian E. Jackson, Kevin Tracy, and Zachary Manchester. "Planning With Attitude". In: *IEEE Robotics and Automation Letters* 6.3 (July 2021), pp. 5658–5664. ISSN: 2377-3766, 2377-3774. DOI: 10.1109/LRA.2021.3052431. (Visited on 02/17/2026).

[67] Riccardo Busetto, Elia Cereda, Marco Forgione, Gabriele Maroni, Dario Piga, and Daniele Palossi. *Nonlinear System Identification Nano-drone Benchmark*. Dec. 2025. DOI: 10.48550/arXiv.2512.14450.

[68] Mato Baotic, Francesco Borrelli, Alberto Bemporad, and Manfred Morari. "Efficient On-Line Computation of Constrained Optimal Control". In: *SIAM J. Control. Optim.* 47.5 (2008), pp. 2470–2489. DOI: 10.1137/060659314.

[69] "Zynq UltraScale+ Device Technical Reference Manual". In: (2025).

[70] *iCE40 UltraPlus Datasheet*. (Visited on 03/11/2026).

[71] National Semiconductor. *PC16550D Universal Asynchronous Receiver/Transmitter with Fifos*. Tech. rep. 1995.

[72] NXP Semiconductors. *UM10204 I2C-bus Specification and User Manual*. Tech. rep. 2021.

[73] *SPI Motorola | PDF*. https://www.scribd.com/document/886578930/SPI-Motorola. (Visited on 03/11/2026).

[74] *nRF52840 Product Specification*. (Visited on 03/11/2026).

[75] *PCB Prototype & PCB Fabrication Manufacturer - JLCPCB*. https://jlcpcb.com/. (Visited on 03/11/2026).

[76] STMicroelectronics. *STM32F405xx and Stm32f407xx Advanced Arm-Based 32-Bit Mcus Datasheet*. June 2018. (Visited on 03/02/2026).

[77] Hengtan Zhang, Zeyu Wang, Jinqiao Yang, Yifu Liang, Yuhan He, Li Gong, Li-Rong Zheng, and Zhuo Zou. “A Dual-Mode Neuromorphic Controller With On-Chip Learning for Robot Motion Control”. In: *IEEE Transactions on Circuits and Systems II: Express Briefs* 72.12 (Dec. 2025), pp. 1872–1876. ISSN: 1558-3791. DOI: 10.1109/TCSII.2025.3577678. (Visited on 03/10/2026).

[78] Raz Halaly and Elishai Ezra Tsur. “Continuous Adaptive Nonlinear Model Predictive Control Using Spiking Neural Networks and Real-Time Learning”. In: *Neuromorphic Computing and Engineering* 4.2 (May 2024), p. 024006. ISSN: 2634-4386. DOI: 10.1088/2634-4386/ad4209. (Visited on 03/02/2026).

[79] Bradley H. Theilman and James B. Aimone. “Solving Sparse Finite Element Problems on Neuromorphic Hardware”. In: *Nature Machine Intelligence* 7.11 (Nov. 2025), pp. 1845–1857. ISSN: 2522-5839. DOI: 10.1038/s42256-025-01143-2. (Visited on 02/08/2026).

[80] Arun L. Bishop, John Z. Zhang, Swaminathan Gurumurthy, Kevin Tracy, and Zachary Manchester. “ReLU-QP: A GPU-Accelerated Quadratic Programming Solver for Model-Predictive Control”. In: *2024 IEEE International Conference on Robotics and Automation (ICRA)*. May 2024, pp. 13285–13292. DOI: 10.1109/ICRA57147.2024.10611249. (Visited on 03/10/2026).

[81] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. “Deep Learning with Limited Numerical Precision”. In: *Proceedings of the 32nd International Conference on Machine Learning*. PMLR, June 2015, pp. 1737–1746. (Visited on 02/08/2026).

[82] Song Han, Huizi Mao, and William J. Dally. “Deep Compression: Compressing Deep Neural Network with Pruning, Trained Quantization and Huffman Coding”. In: *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*. Ed. by Yoshua Bengio and Yann LeCun. 2016. (Visited on 02/09/2026).

[83] Scott Kuindersma, Robin Deits, Maurice Fallon, Andrés Valenzuela, Hongkai Dai, Frank Permenter, Twan Koolen, Pat Marion, and Russ Tedrake. “Optimization-Based Locomotion Planning, Estimation, and Control Design for the Atlas Humanoid Robot”. In: *Autonomous Robots* 40.3 (Mar. 2016), pp. 429–455. ISSN: 0929-5593, 1573-7527. DOI: 10.1007/s10514-015-9479-3. (Visited on 03/10/2026).

[84] Arun L. Bishop, Juan Alvarez-Padilla, Sam Schoedel, Ibrahima Sory Sow, Juee Chandrachud, Sheitej Sharma, Will Kraus, Beomyeong Park, Robert J. Griffin, John M. Dolan, and Zachary Manchester. “The Surprising Effectiveness of Linear Models for Whole-Body Model-Predictive Control”. In: *24th IEEE-RAS International Conference on Humanoid Robots, Humanoids 2025, Seoul, Republic of Korea, September 30 - October 2, 2025*. IEEE, 2025, pp. 1–7. DOI: 10.1109/HUMANOIDS65713.2025.11203045. (Visited on 02/08/2026).