FPGA-MPC: a FPGA-accelerated ADMM Solver for Convex Model Predictive Control

Design Document

Alexander Du (asd2192), Apurva Reddy (akr2177),
Roy Hwang (rjh2173), Godwill Agbehonou (gea2118)

Spring 2025

1) Introduction

Model Predictive Control (MPC) is a feedback control strategy that has seen great success in many robotic applications [1, 2]. MPC often leverages trajectory optimization (TO) on an objective function, subject to system dynamics and other constraints. These TO problems have hundreds/thousands of variables and must be solved in milliseconds to achieve real-time performance. As such, careful approximations/simplifications of underlying numerical methods and the optimal control problem are often used to enable real-time deployments on a variety of hardware platforms [3].

This project, **FPGA-MPC**, is a fast quadratic programming solver for MPC, which is inspired by recent works that leverage acceleration on GPUs and FPGAs for online trajectory optimization [4, 5, 6, 7]. In particular, *TinyMPC* [8] exploits the closed-form solution of LQR through Riccati recursion and compresses the ADMM algorithm to enable high frequency control on resource-constrained platforms. In this work we aim to accelerate *TinyMPC* on an FPGA, unlocking better performance and solver capabilities, and ultimately to demonstrate **FPGA-MPC** on a simulated + real quadrotor.

2) System Design

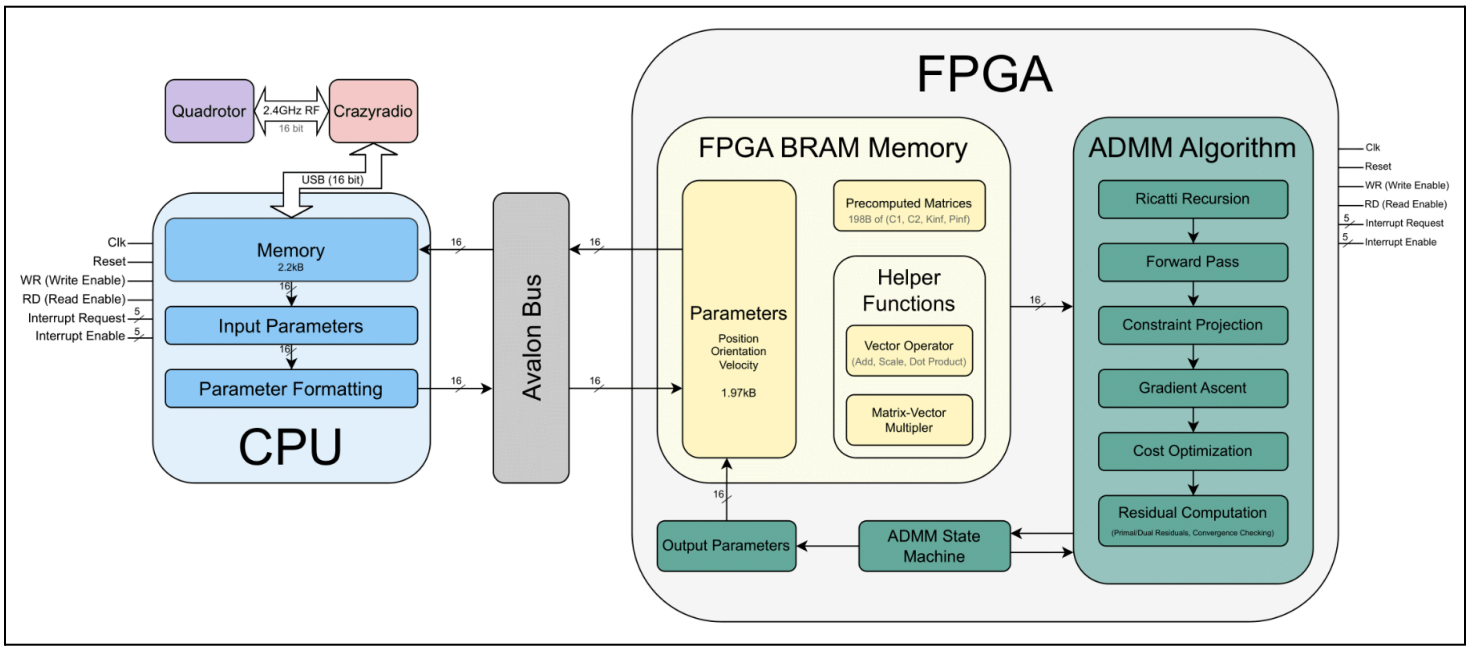

Figure 1: System Block Diagram of FPGA-MPC

The quadrotor/drone knows its current position but relies on external instructions on where to go next. It sends its current position and orientation data as a **16-bit struct** via 2.4GHz Radio to the Crazyradio. Crazyradio forwards the data over via USB to an external CPU. Assuming Write Enable is on, the CPU stores the incoming parameters in memory. The CPU processes the drone's position and orientation data to compute the drone's linear velocity, angular velocity, and other important parameters. It converts this raw parameter data into a format that is friendly for the FPGA to read. The CPU writes this formatted parameter data via an Avalon Bus interface into the FPGA's Block Random Access Memory (BRAM).

The FPGA's job is to calculate the drone's next position and orientation given the drone's current position and orientation. It does so using the Alternating Direction Method of Multipliers (ADMM) algorithm, which will be described further in the following section. In order to reduce the number of computations the FPGA has to do, precomputed matrices from the infinite horizon solution to the LQR problem are cached in the FPGA's memory. Helper functions simplify future calculations. After the ADMM Algorithm finishes, the FPGA outputs a new set of controls in a 16-bit struct to its memory. This data is written back to the CPU memory through the Avalon Bus. The CPU forwards the new control actions via USB to the Crazyradio, which subsequently forwards them to the quadrotor via radio.

Figure 2: Crazyflie 2.1 - Open Source Mirco Quadcopter Drone

Figure 3: Crazyradio PA 2.4 GHz USB dongle

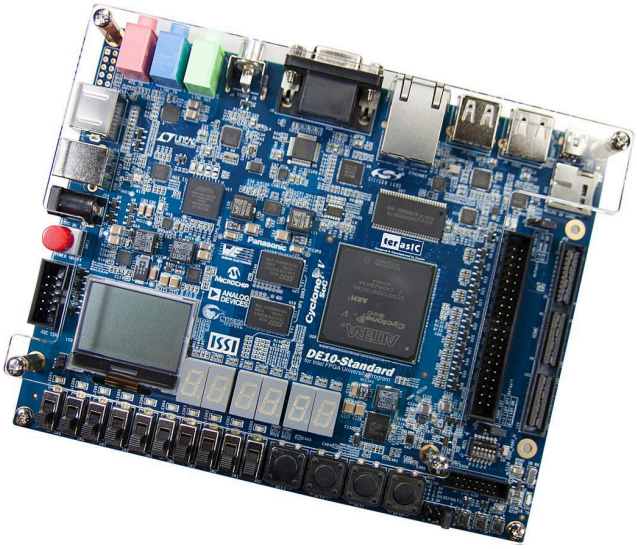

Figure 4: DE1-SoC FPGA

3) Algorithm

TinyMPC's Simplified Alternating Direction Method of Multipliers (ADMM) Algorithm

Python

```python
def solve_admm(self, x_init, u_init, x_ref=None, u_ref=None):
    x, u = np.copy(x_init), np.copy(u_init)

    # x_ref and u_ref can be passed in for trajectory following, otherwise use zero reference
    x_ref = np.zeros(x.shape) if x_ref is None else x_ref
    u_ref = np.zeros(u.shape) if u_ref is None else u_ref

    # Initialize variables from previous solve
    v, z = np.copy(self.v_prev), np.copy(self.z_prev)
    g, y = np.copy(self.g_prev), np.copy(self.y_prev)
    q = np.copy(self.q_prev)

    # Keep track of previous values for residuals
    v_prev, z_prev = np.copy(v), np.copy(z)

    r = np.zeros(u.shape) # control input residual
    p = np.zeros(x.shape) # linear term of cost-to-go (value) function
    d = np.zeros(u.shape) # feedforward term

    for _ in range(self.max_iter):
        # ----- Primal update (to get x' and u') -----
        # d: feedforward term
        # p: linear term of cost-to-go (value) function

        # Solve LQR problem with Riccati recursion
        # Riccati matrices (C1, C2) and Kinf are precomputed and cached, so here we only update linear terms
        # C1 = (R + B.T @ Pinf @ B)^-1
        # C2 = (A - B @ Kinf).T
        # Kinf: infinite horizon gain
        for k in range(self.N-2, -1, -1):
            d[:, k] = np.dot(self.cache['C1'], np.dot(self.cache['B'].T, p[:, k + 1]) + r[:, k])
            p[:, k] = q[:, k] + np.dot(self.cache['C2'], p[:, k + 1]) - np.dot(self.cache['Kinf'].T, r[:, k])

        # Forward pass to rollout trajectory (u_0, x_1, u_1, x_2, ..., u_N-1, x_N)
        for k in range(self.N - 1):
            # Compute control (u_k) with feedback controller gain (K_inf) and feedforward term (d)
            u[:, k] = -np.dot(self.cache['Kinf'], x[:, k]) - d[:, k]
            # Transition to next state (x_k+1) with control (uk) and linearized dynamics (A, B)
            x[:, k + 1] = np.dot(self.cache['A'], x[:, k]) + np.dot(self.cache['B'], u[:, k])

        # ------------------------------------------------------------
        # ----- Slack update (to get z' and v') -----

        # Linear projection of updated state and control onto their feasible sets
        # Used to enforce constraints
        # Note: y and g are swapped in the TinyMPC paper
        for k in range(self.N - 1):
            z[:, k] = np.clip(u[:, k] + y[:, k], self.umin, self.umax)
            v[:, k] = np.clip(x[:, k] + g[:, k], self.xmin, self.xmax)
        v[:, self.N-1] = np.clip(x[:, self.N-1] + g[:, self.N-1], self.xmin, self.xmax)

        # ------------------------------------------------------------
        # ----- Dual update (to get y' and g') -----
        # Adjust dual variables (lagrange multipliers) to penalize mismatch between x and v, u and z
        # - Gradient ascent on the dual variables
        # y, g: scaled dual variables (lambda_k / rho, mu_k / rho)
```

```
        for k in range(self.N - 1):
            y[:, k] += u[:, k] - z[:, k]
            g[:, k] += x[:, k] - v[:, k]
        g[:, self.N-1] += x[:, self.N-1] - v[:, self.N-1]


        # ---------------------------------------------------------
        # ----- Linear cost update (to get r', q', p') -----
        # Update vectors used in next iteration's Riccati recursion:
        # q, r: state and control input residuals
        # p: terminal cost residual
        for k in range(self.N - 1):
            r[:, k] = -self.cache['R'] @ u_ref[:, k]
            r[:, k] -= self.cache['rho'] * (z[:, k] - y[:, k])

            q[:, k] = -self.cache['Q'] @ x_ref[:, k]
            q[:, k] -= self.cache['rho'] * (v[:, k] - g[:, k])

        p[:,self.N-1] = -np.dot(self.cache['Pinf'], x_ref[:, self.N-1])
        p[:,self.N-1] -= self.cache['rho'] * (v[:, self.N-1] - g[:, self.N-1])

        # ---------------------------------------------------------
        # ----- Compute residuals and check convergence -----
        pri_res_input = np.max(np.abs(u - z))
        pri_res_state = np.max(np.abs(x - v))
        dua_res_input = np.max(np.abs(self.cache['rho'] * (z_prev - z)))
        dua_res_state = np.max(np.abs(self.cache['rho'] * (v_prev - v)))

        z_prev = np.copy(z)
        v_prev = np.copy(v)

        # Exit condition (if all residuals are below tolerance)
        if (pri_res_input < self.abs_pri_tol and dua_res_input < self.abs_dua_tol and
            pri_res_state < self.abs_pri_tol and dua_res_state < self.abs_dua_tol):
            break

        # ---------------------------------------------------------

    # Save variables for next solve
    self.x_prev, self.u_prev = x, u
    self.v_prev, self.z_prev = v, z
    self.g_prev, self.y_prev = g, y
    self.q_prev = q

    return x, u
```

Rest of TinyMPC's solver

Python

```
def __init__(self, A, B, Q, R, Nsteps, rho=1.0, n_dlqr_steps=500, mode = 'hover'):
    """Initialize TinyMPC with direct system matrices and compute DLQR automatically

    Args:
        A (np.ndarray): System dynamics matrix
        B (np.ndarray): Input matrix
        Q (np.ndarray): State cost matrix
        R (np.ndarray): Input cost matrix
```

```
        Nsteps (int): Horizon length
        rho (float): Initial rho value
        n_dlqr_steps (int): Number of steps for DLQR computation
    """
    # Get dimensions
    self.nx, self.nu = A.shape[0], B.shape[1]
    self.N = Nsteps
    self.mode = mode # hover or trajectory following

    # set tolerances and iterations based on mode
    if self.mode == 'hover':
        self.max_iter = 500
        self.abs_pri_tol = 1e-2
        self.abs_dua_tol = 1e-2
    else:
        self.max_iter = 10
        self.abs_pri_tol = 1e-3
        self.abs_dua_tol = 1e-3


    # Compute DLQR solution for terminal cost
    P_lqr = self._compute_dlqr(A, B, Q, R, n_dlqr_steps)

    # Initialize cache with computed values
    self.cache = {
        'rho': rho,
        'A': A,
        'B': B,
        'Q': P_lqr,  # Use DLQR solution for terminal cost
        'R': R
    }

    # Initialize state variables
    self.v_prev, self.z_prev = np.zeros((self.nx, self.N)), np.zeros((self.nu, self.N-1))
    self.g_prev, self.y_prev = np.zeros((self.nx, self.N)), np.zeros((self.nu, self.N-1))
    self.q_prev = np.zeros((self.nx, self.N))

    # Initialize previous solutions for warm start
    self.x_prev, self.u_prev = np.zeros((self.nx, self.N)), np.zeros((self.nu, self.N-1))

    self.Q, self.R = Q, R

    # Compute cache terms (Kinf, Pinf, C1, C2)
    self.compute_cache_terms()

def _compute_dlqr(self, A, B, Q, R, n_steps):
    """Compute Discrete-time LQR solution"""
    P = Q
    for _ in range(n_steps):
        #K = np.linalg.inv(R + B.T @ P @ B) @ B.T @ P @ A
        K = np.linalg.solve(
            R + B.T @ P @ B + 1e-8*np.eye(B.shape[1]),  # Add regularization
            B.T @ P @ A
        )
        P = Q + A.T @ P @ (A - B @ K)
    return P


def compute_cache_terms(self):
    """Compute and cache terms for ADMM"""
    Q_rho, R_rho = self.cache['Q'], self.cache['R']
    R_rho += self.cache['rho'] * np.eye(R_rho.shape[0])
    Q_rho += self.cache['rho'] * np.eye(Q_rho.shape[0])

    A, B = self.cache['A'], self.cache['B']
    Kinf = np.zeros(B.T.shape)
```

```
Pinf = np.copy(self.cache['Q'])

# Compute infinite horizon solution (Kinf, Pinf)
for k in range(5000):
    Kinf_prev = np.copy(Kinf)
    Kinf = np.linalg.inv(R_rho + B.T @ Pinf @ B) @ B.T @ Pinf @ A
    Pinf = Q_rho + A.T @ Pinf @ (A - B @ Kinf)

    if np.linalg.norm(Kinf - Kinf_prev, 2) < 1e-10:
        break

# Compute Riccati matrices
AmBKt = (A - B @ Kinf).T
Quu_inv = np.linalg.inv(R_rho + B.T @ Pinf @ B)

print(Kinf.shape) # (nu, nx), since u_k_* = -Kinf @ x_k - d_k
print(Pinf.shape) # (nx, nx)
print(Quu_inv.shape) # (nu, nu)
print(AmBKt.shape) # (nx, nx)

# Cache computed terms
self.cache['Kinf'] = Kinf
self.cache['Pinf'] = Pinf
self.cache['C1'] = Quu_inv
self.cache['C2'] = AmBKt
```

Overview:

The key things that the algorithm handles with the drone can be divided into three parts. (1) What the drone behaves like, (2) what it should care about, and (3) how far into the future it should plan.

Matrices A and B are the system dynamic matrices essentially describe how the drone has evolved over time. Matrices Q and R are called cost matrices and they are what the drone cares about. Q represents cost on the state which describes position error, velocity, error, angle deviation. R is the cost of the control effort which essentially describes how much "control" we should use which could be related to thrust, torque, etc. N-steps is how many steps into the future to plan. Rho is a tuning parameter to set state/control constraints on the ADMM solver. N_dlqr_steps is how many iterations we are running to compute the terminal cost matrix.

Initialization:

After the matrices are inputted, the DLQR solution is used to compute the terminal cost matrix to stabilize the controller. Buffers are initialized to store previous values of state and control to help the solver.

Precomputed Matrices:

Kin, C1, and C2 are all precomputed matrices used to make the forward-backward passes extremely fast. Kin is a steady-state LQR Gain. C1 is an inverse matrix used in Riccati Recursion. C2 is part of the backward pass recursion formula.

Compute DLQR:

This module essentially iteratively solves the Riccati equation to compute the terminal cost matrix for DLQR. It ensures smooth stability at the end of the horizon.

Solve ADMM:

The Solve _admm function is the core solver that essentially runs at each control step. In the backward pass, we are commuting the feedforward control term and the cost-to-go vector. We are essentially minimizing the current total cost knowing how costly the future is.

In the forward pass, we simulated what the control and state will be by using the feedback controller and system dynamics. It computes optimal control at step k using the current state and precomputed d. We apply the system model to get to the next state.

This model essentially simulates the planned trajectory using the optimal control law from the backward pass. After, we project u and x onto thrust bounds, we update our dual variables to penalize deviations from constraints, and update our linear cost terms for the next iteration.

The solver stops early if both primal and dual residuals are small enough.

Output:

The output of TinyMPC is the full predicted state and control trajectories. The drone will only use the first control this input and re-run the whole function on the next time step.

4) Resources

We will primarily be using the FPGA for storing the various matrices we are using, multiplying our matrices using (DSP blocks). As a disclaimer, exact dimensions of the matrices have not been specifically determined so we will use estimated dimensions.

Matrix Storage:

We assume Fixed-Point 16 Bit Entry, so each matrix element is 2 bytes. To interpret this table, consider this sample calculation for A: 6x6 elements X 2 bytes per element = 72 bytes

Matrix	Size	Elements	Bytes
`A`	6×6	36	72 B
`B`	6×3	18	36 B
`Q`	6×6	36	72 B
`R`	3×3	9	18 B
`Kinf`	3×6 (gain matrix)	18	36 B
`Pinf`	6×6	36	72 B
`C1`	3×3	9	18 B
`C2`	6×6	36	72 B

Subtotal: 400 B = **0.4 KB**

Trajectory Matrices:

Variable	Size	Elements	Bytes (16-bit)
`x`	N_x X N	6×20 = 120	240 B
`u`	N_u X (N_1)	3×19 = 57	114 B
`v, g, p, q`	same as `x`	4 × 240 B	960 B
`z, y, d, r`	same as `u`	4 × 114 B	456 B

Subtotal = **1.77 KB**

Thus, our total On-Chip Memory will be approximately **2.2 KB** which falls well below the limit of 512 KB on the DE1.

Matrix Multipliers:

We will be performing several parallel 6x6 matrix operations for different operations like Riccati backward updates. The DE1 has 112 DSP blocks which is plenty for our use case.

Controller FSM/Logic:

This includes Backward Pass FSM, Forward Pass FSM, ADMM Iterations Controller. We won't need a large amount of LUTs and Flip-Flops that overflow the DE1 for this project.

External Memory:

We plan to keep our horizon and state small enough so that TinyMPC can stay entirely on-chip because accessing off-chip memory can be very slow. Thus we will attempt to solely use BRAM on the FPGA.

5) Hardware/Software Interface

Interfacing with the SystemVerilog code, the algorithm logic running in the Linux kernel module would handle preparing and sending the small, fixed-size matrices (representing the MPC state and dynamics) to the FPGA accelerator. Since FPGA-MPC is designed to operate entirely on-chip, the kernel module facilitates tight coupling between the CPU and the FPGA by memory-mapping the newly created mat-mul hardware registers and exposing a lightweight interface to the MPC solver. The module writes the A and B matrices to the hardware, triggers computation via a control register, and reads back the resulting C matrix once the operation is complete. This enables efficient offloading of critical linear algebra operations—such as matrix multiplications in the MPC update step—while avoiding the latency overhead of accessing external memory, thus preserving the real-time performance guarantees required by FPGA-MPC.

There would be two components to creating the interface: the FPGA image to be created for the on-chip matrix calculations and the kernel module to enable the software and actual testing of the FPGA-MPC algorithm.

Device Tree pseudocode:

```
Unset
matmul-1.0: matmul@0x100000000 {
    compatible = "csee4840,matmul-1.0";
    reg = <0x43C00000 0x1000>;
}; //end matmul@0x100000000 (matmul-1.0)
```

The device driver/module code would be similar to the C code example below (focused on matrix multiplication for the concept code for a 2x2 matrix):

```
C/C++
/*
 * Device driver for the TinyMPC 2x2 Matrix Multiplication Accelerator
 *
 * A Platform device implemented using the misc subsystem
 *
 * Alexander Du (asd2192), Apurva Reddy (akr2177), Roy Hwang (rjh2173), Godwil
 * Agbehonou (gea2118)
 * Columbia University
 */

#include <linux/module.h>
#include <linux/init.h>
#include <linux/errno.h>
#include <linux/version.h>
#include <linux/kernel.h>
#include <linux/platform_device.h>
#include <linux/miscdevice.h>
#include <linux/slab.h>
```

```c
#include <linux/io.h>
#include <linux/of.h>
#include <linux/of_address.h>
#include <linux/fs.h>
#include <linux/uaccess.h>
#include "tinympc_matmul.h"

#define DRIVER_NAME "tinympc_matmul"

// Register offsets for the FPGA accelerator hardware
#define REG_CONTROL   0x00
#define REG_A00       0x04
#define REG_A01       0x08
#define REG_A10       0x0C
#define REG_A11       0x10
#define REG_B00       0x14
#define REG_B01       0x18
#define REG_B10       0x1C
#define REG_B11       0x20
#define REG_C00       0x24
#define REG_C01       0x28
#define REG_C10       0x2C
#define REG_C11       0x30

// Device structure to store hardware resource information
struct matmul_dev {
    struct resource res;         //Memory resource
    void __iomem *virtbase;      //Virtual address base
    int32_t C[4];                        //Resulting matrix C
} dev;

// Initiates matrix multiplication using the FPGA accelerator
static void matmul_start(const struct matmul_matrices *mat)
{
    // Write input Matrix A to hardware registers
    iowrite32(mat->A[0][0], dev.virtbase + REG_A00);
    iowrite32(mat->A[0][1], dev.virtbase + REG_A01);
    iowrite32(mat->A[1][0], dev.virtbase + REG_A10);
    iowrite32(mat->A[1][1], dev.virtbase + REG_A11);
    //Write input matrix B to hardware registers
    iowrite32(mat->B[0][0], dev.virtbase + REG_B00);
    iowrite32(mat->B[0][1], dev.virtbase + REG_B01);
    iowrite32(mat->B[1][0], dev.virtbase + REG_B10);
    iowrite32(mat->B[1][1], dev.virtbase + REG_B11);

    // Computation
    iowrite32(0x1, dev.virtbase + REG_CONTROL);

    // Wait for done bit
    while (!(ioread32(dev.virtbase + REG_CONTROL) & 0x2));

    // Read result from hardware registers into output matrix C
    dev.C[0] = ioread32(dev.virtbase + REG_C00);
    dev.C[1] = ioread32(dev.virtbase + REG_C01);
    dev.C[2] = ioread32(dev.virtbase + REG_C10);
    dev.C[3] = ioread32(dev.virtbase + REG_C11);
}
```

```
// Handles ioctl calls from userspace applications
static long matmul_ioctl(struct file *f, unsigned int cmd, unsigned long arg)
{
    struct matmul_matrices user_mat;
    int32_t user_result[4];

    switch (cmd) {
    case MATMUL_CMD_RUN:
       // Copy matrix data from userspace
        if (copy_from_user(&user_mat, (void __user *)arg, sizeof(user_mat)))
            return -EFAULT;

        // Perform matrix multiplication
        matmul_start(&user_mat);

       // Copy results back to userspace
        memcpy(user_result, dev.C, sizeof(user_result));
        if (copy_to_user((void __user *)arg, user_result, sizeof(user_result)))
            return -EFAULT;
        break;

    default:
        return -EINVAL;
    }

    return 0;
}

// File operations struct
static const struct file_operations matmul_fops = {
    .owner = THIS_MODULE,
    .unlocked_ioctl = matmul_ioctl,
};

// Miscellaneous device registration struct
static struct miscdevice matmul_misc_device = {
    .minor = MISC_DYNAMIC_MINOR,
    .name = DRIVER_NAME,
    .fops = &matmul_fops,
};

// Device probe function called when device is found
static int __init matmul_probe(struct platform_device *pdev)
{
    int ret;

    ret = misc_register(&matmul_misc_device);
    if (ret)
        return ret;

   // Get hardware resource from device tree
    ret = of_address_to_resource(pdev->dev.of_node, 0, &dev.res);
    if (ret)
        goto fail_deregister;

   // Request memory region for hardware
    if (!request_mem_region(dev.res.start, resource_size(&dev.res), DRIVER_NAME)) {
```

```
        ret = -EBUSY;
        goto fail_deregister;
    }

    // Map hardware address to kernel virtual address space
    dev.virtbase = of_iomap(pdev->dev.of_node, 0);
    if (!dev.virtbase) {
        ret = -ENOMEM;
        goto fail_release;
    }

    pr_info(DRIVER_NAME ": FPGA matmul driver loaded\n");
    return 0;

fail_release:
    release_mem_region(dev.res.start, resource_size(&dev.res));
fail_deregister:
    misc_deregister(&matmul_misc_device);
    return ret;
}

// Cleanup when device is removed
static int matmul_remove(struct platform_device *pdev)
{
    iounmap(dev.virtbase);
    release_mem_region(dev.res.start, resource_size(&dev.res));
    misc_deregister(&matmul_misc_device);
    pr_info(DRIVER_NAME ": FPGA matmul driver removed\n");
    return 0;
}

#ifdef CONFIG_OF
// Compatible devices for device tree
static const struct of_device_id matmul_of_match[] = {
    { .compatible = "tinympc,matmul-1.0" },
    {},
};
MODULE_DEVICE_TABLE(of, matmul_of_match);
#endif

// Platform driver definition
static struct platform_driver matmul_driver = {
    .driver = {
        .name = DRIVER_NAME,
        .of_match_table = of_match_ptr(matmul_of_match),
    },
    .remove = __exit_p(matmul_remove),
};

static int __init matmul_init(void)
{
    pr_info(DRIVER_NAME ": init\n");
    return platform_driver_probe(&matmul_driver, matmul_probe);
}

static void __exit matmul_exit(void)
{
```

```
    platform_driver_unregister(&matmul_driver);
    pr_info(DRIVER_NAME ": exit\n");
}

module_init(matmul_init);
module_exit(matmul_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Godwill Agbehonou");
MODULE_DESCRIPTION("TinyMPC 2x2 Matrix Multiplication FPGA Driver");
```

The accompanying .h file

C/C++

```
#ifndef TINY_MPC_MATMUL_H
#define TINY_MPC_MATMUL_H

#include <linux/ioctl.h>

typedef struct {
        int32_t A[2][2];
        int32_t B[2][2];
} matmul_matrices;

#define MATMUL_CMD_RUN _IOWR('m', 1, matmul_matrices)

#endif // TINY_MPC_MATMUL_H
```

Example Code Usage:

C/C++

```
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/ioctl.h>
#include "tinympc_matmul.h"

int main() {
        int fd = open("/dev/tinympc_matmul", O_RDWR);
        matmul_matrices m = {{{1, 2}, {3, 4}}, {{5, 6}, {7, 8}}};

        if (ioctl(fd, MATMUL_CMD_RUN, &m) == 0) {
                printf("Result:\n");
                printf("[%d %d]\n[%d %d]\n", m.A[0][0], m.A[0][1], m.A[1][0], m.A[1][1]);
        } else {
                perror("ioctl failed");
        }
        close(fd);
        return 0;
}
```

References

[1] F.R.Hogan,E.R.Grau,andA.Rodriguez,"Reactiveplanarmanipula- tion with convex hybrid mpc," in 2018 IEEE International Conference on Robotics and Automation (ICRA). IEEE, 2018, pp. 247–253.

[2] P. M. Wensing, M. Posa, Y. Hu, A. Escande, N. Mansard, and A. Del Prete, "Optimization-based control for dynamic legged robots," IEEE Transactions on Robotics, 2023.

[3] S. Kuindersma, R. Deits, M. Fallon, A. Valenzuela, H. Dai, F. Per- menter, T. Koolen, P. Marion, and R. Tedrake, "Optimization-based locomotion planning, estimation, and control design for the atlas humanoid robot," Autonomous robots, vol. 40, pp. 429–455, 2016.

[4] T. Antony and M. J. Grant, "Rapid Indirect Trajectory Optimization on Highly Parallel Computing Architectures," vol. 54, no. 5, pp. 1081– 1091.

[5] B. Plancher, S. M. Neuman, R. Ghosal, S. Kuindersma, and V. J. Reddi, "GRiD: GPU-Accelerated Rigid Body Dynamics with Analytical Gradients," in 2022 International Conference on Robotics and Automation (ICRA). IEEE, pp. 6253–6260.

[6] Y. Lee, M. Cho, and K.-S. Kim, "Gpu-parallelized iterative lqr with input constraints for fast collision avoidance of autonomous vehicles," in 2022 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), 2022, pp. 4797–4804.

[7] E. Adabag, M. Atal, W. Gerard, and B. Plancher, "Mpcgpu: Real-time nonlinear model predictive control through preconditioned conjugate gradient on the gpu," in 2024 IEEE International Conference on Robotics and Automation (ICRA). IEEE, 2024, pp. 9787–9794.

[8] K. Nguyen, S. Schoedel, A. Alavilli, B. Plancher, and Z. Manchester, "Tinympc: Model-predictive control on resource-constrained micro- controllers," in 2024 IEEE International Conference on Robotics and Automation (ICRA). IEEE, 2024, pp. 1–7.