MPC Solver Hardware Generation Framework with Model-Specific Operation Fusion and Pruning

Zhenyu Wu*†, Brian Plancher‡, Ian McInerney§, Hayden Kwok-Hay So†, Maolin Wang**††, and Kwang-Ting Cheng**
*AI Chip Center for Emerging Smart Systems
†The University of Hong Kong
‡Dartmouth College and Barnard College, Columbia University
§Imperial College London
**The Hong Kong University of Science and Technology
{zhenyuwu, hso}@eee.hku.hk, plancher@dartmouth.edu, i.mcinerney17@imperial.ac.uk, {maolinwang, timcheng}@ust.hk

***Abstract*—Model Predictive Control (MPC) is a state-of-the-art and robust control framework. However, its stringent performance requirements for computational infrastructure, and its need for real-time computation at the edge, hinder its widespread adoption in various application scenarios. This is particularly challenging for the MCUs commonly found on tiny robots. To address this computational challenge, we developed a flexible MPC solver hardware generation framework which includes a parameterized and programmable vector architecture template that accommodates instruction-level and data-level parallelism in vector and matrix functional units, and a model-specific fused architecture. Implementation of the proposed processor on the Ultra96 platform achieves up to a $9.73\times$ speedup compared to existing generic solutions on MCUs. Moreover, end-to-end performance tests reveal that this speedup reduces the overall control error by 25.96%. Overall, the enhanced flexibility and performance of our proposed processor design open up the potential for MPC to be utilized in a broader range of applications.**

I. Introduction

Model Predictive Control (MPC) is an advanced control technique widely used in industries such as process control, robotics, aviation, and electrical grids due to its ability to handle multi-variable systems, incorporate constraints, and optimize performance over a future horizon [6], [7], [16], [28], [32], [33]. MPC repeatedly solves an underlying trajectory optimization [2] problem over an N-step horizon. This problem minimizes a cost function J over the state $x_k \in \mathbb{R}^n$ and control $u_k \in \mathbb{R}^m$ variables at each time step k. The system evolves according to a dynamics function f and the states and controls are constrained within sets $\mathcal{X}$ and $\mathcal{U}$. As such, MPC repeatedly solves the following optimization problem at each control step, where $\bar{x}$ is the current system state estimate:

$$\begin{aligned} \min_{x_{1:N}, u_{1:N-1}} \quad & J(x_{1:N}, u_{1:N-1}) \\ \text{s.t.} \quad & x_{k+1} = f(x_k, u_k), \quad k = 1, \ldots, N-1 \\ & x_k \in \mathcal{X}, \; u_k \in \mathcal{U}, \quad k = 1, \ldots, N \\ & x_0 = \bar{x}. \end{aligned} \tag{1}$$

This research was conducted by ACCESS – AI Chip Center for Emerging Smart Systems, supported by the InnoHK initiative of the Innovation and Technology Commission of the Hong Kong Special Administrative Region Government. ††Corresponding author.

The first control action u_1, from the solution of (1), is applied to the system which then evolves according to its real-world dynamics. The process then repeats, with a new sensor reading updating $\bar{x}$, followed by the solution of (1) again.

While a very powerful technique, the widespread application of MPC is often hindered by its significant computational challenges [25]. Take a trajectory tracking task of a quadrotor as an example, while a longer prediction horizon N and higher control update rate can significantly improve the control performance as shown in Figure 1, it can also significantly increase the computational complexity of the controller. As such, to be used for real-time embedded control, MPC requires a low-latency and high-throughput real-time optimization solver on embedded platforms. Balancing the trade-off of control performance and computational efficiency remains a critical challenge in deploying MPC across various domains [29].

Recent advances in cached MPC algorithms [3], [21], [26] have paved the way for more efficient control systems, demonstrating high-performance on physical robot hardware. Further adaptation of these algorithms for reconfigurable platforms, such as FPGAs, can offer more power-efficient and faster MPC solutions, improving real-world performance. However, the wide variety of available FPGA platforms and differing performance requirements present several challenges in the mapping process. Designers must carefully select platforms based on a balance between specific power consumption and speed requirements. Therefore, a versatile design framework that supports multiple target platforms and provides accurate performance projections is highly desirable. In this work, we propose such a novel design framework with the following key contributions:

- We propose the first open-source, high-performance FPGA solver generation framework for control problems, based on recent, state-of-the-art, cached MPC algorithms, enabling agile deployment of embedded controllers.
- We employ the built-in scheduler of HLS tools to automatically extract instruction-level parallelism from sequentially dependent operations in the generated solver.
- We utilize model-specific sparsity-aware processing to eliminate redundant computations and improve resource

 DOI: 10.1109/ICFPT67023.2025.00047

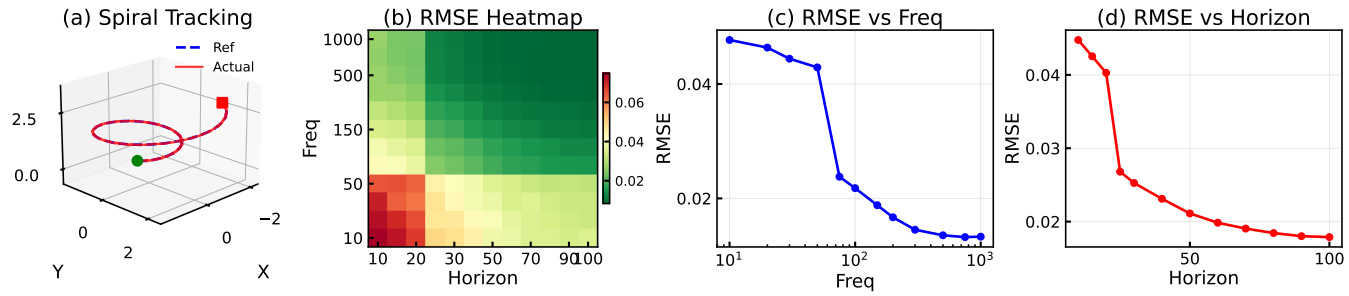

Fig. 1. Effect of different prediction horizon and control update rate when executing a quadrotor trajectory tracking task in terms of the Root Mean Square Error (RMSE) between the reference trajectory and actual trajectory. Longer prediction horizons and higher control update rates bring better control performance.

efficiency.

The paper is organized as follows: Section II introduces details about our target cached MPC algorithm and its architecture design challenge. Section III and Section IV introduce two methods we explored to generate hardware solvers. Section V evaluates the performance of our proposed solutions on an FPGA. Section VI summarizes our work. Our implementation code is publicly available at https://github.com/KevinLikesDringCoffe/tinympc_ip_gen.

II. Background

A. Related Work

Researchers have been actively exploring FPGA designs to provide real-time power-efficient implementations of Model Predictive Control (MPC) in many application areas such as power electronics [20], [31], unmanned aerial vehicles [5], fusion reactors [8], aviation [9] and vehicles [18].

Various algorithms and architectures have been proposed for implementing MPC on FPGAs [23]. Initial work focused on implementing interior-point Quadratic Programming (QP) solvers on FPGAs [9], [10], [12], [13], with designs proposing various optimizations such as tailored MINRES solvers [9], [12], compressed diagonal storage for the matrices [13], or a systolic array Cholesky factorization [19]. Further works proposed using active-set QP solvers, which solve a sequence of equality-constrained QP subproblems, based on Givens rotations and the QR factorization [15], [34] or direct matrix inversion [35]. Other works proposed using gradient-based methods, such as the Fast Gradient Method or the Alternating Direction Method of Multipliers (ADMM), achieving Megahertz-rate controllers on simple MPC problems by utilizing highly-pipelined matrix-vector multiplications [11], [14], [27]. Importantly, we note that there have been no systematic investigation into designing modern and efficient architectures for recent advances in traditional optimization algorithms for MPC, such as TinyMPC [26], which reduce solver algorithm complexity.

Finally, recently, researchers have proposed training machine learning models to approximate the MPC control law offline and then implementing the closed-loop control via ML model inference on the FPGA [5], [20], [31]. While the ML-based approaches appear promising for implementing MPC, they sacrifice constraint satisfaction and safety guarantees for faster control rates.

B. The Computation Flow of TinyMPC

The general MPC problem (1) is often solved approximately, where the cost is modeled quadratically and the dynamics and set constraints are modeled linearly. In particular, in many robotic applications, $J = \frac{1}{2}x_N^T Q_f x_N + q_f^T x_N + \sum_{k=1}^{N-1} \frac{1}{2}x_k^T Q x_k + q_k^T x_k + \frac{1}{2}u_k^T R u_k + r_k^T u_k$, and $f = Ax_k + Bu_k$. This produces a QP, whose solution is a well-researched area with many known fast CPU-based solvers (e.g., OSQP [30], DAQP [1], CVXGEN [22]).

```
def solve():
    for _ in range(max_iter):
        # 1. Forward pass
        for i in range(N-1):
            u[i] = -Kinf @ x[i] - d[i]
            x[i+1] = A @ x[i] + B @ u[i]

        # 2. Projection (box constraints)
        z = [min(u_max, max(u_min, u[i]+y[i])) for
            i in range(N-1)]
        v = [min(x_max, max(x_min, x[i]+g[i])) for
            i in range(N)]

        # 3. Dual update
        y = [y[i] + u[i] - z[i] for i in range(N-1)]
        g = [g[i] + x[i] - v[i] for i in range(N)]

        # 4. Cost terms
        r = [-Uref[i]*R - rho*(z[i]-y[i]) for i in
            range(N-1)]
        q = [-Xref[i]*Q - rho*(v[i]-g[i]) for i in
            range(N)]
        p[-1] = -Pinf.T@Xref[-1] - rho*(v[-1]-g[-1])

        # 5. Backward pass
        for i in range(N-2, -1, -1):
            d[i] = C1 @ (B.T @ p[i+1] + r[i])
            p[i] = q[i] + C2 @ p[i+1] - Kinf.T@r[i]
```

Fig. 2. Pseudo-code description of the TinyMPC algorithm. The five key steps are: (1) forward pass, (2) constraint projection, (3) dual variable update, (4) linear cost computation, and (5) backward pass.

TinyMPC [26] further approximates this problem for more efficient online computation by splitting, and iteratively solving, the set constraints from the rest of the problem through an algorithmic technique known as the Alternating Direction

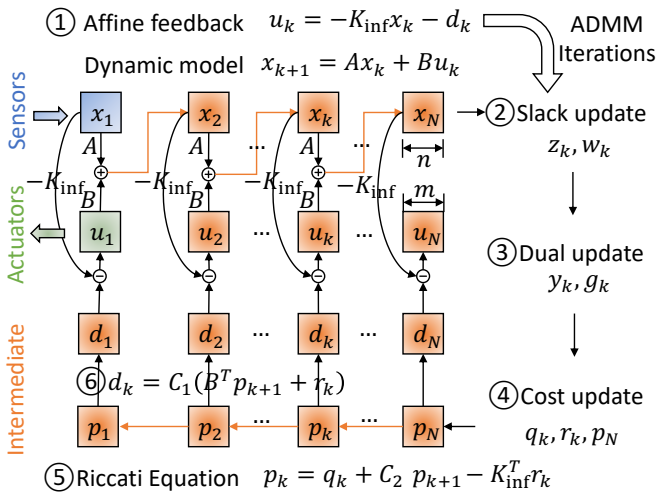

Fig. 3. TinyMPC computation flow. The arrows in the figure represents computation dependency. $K_{\text{inf}} \in \mathbb{R}^{m\times n}, C_1 \in \mathbb{R}^{m\times m}, C_2 \in \mathbb{R}^{n\times n}$

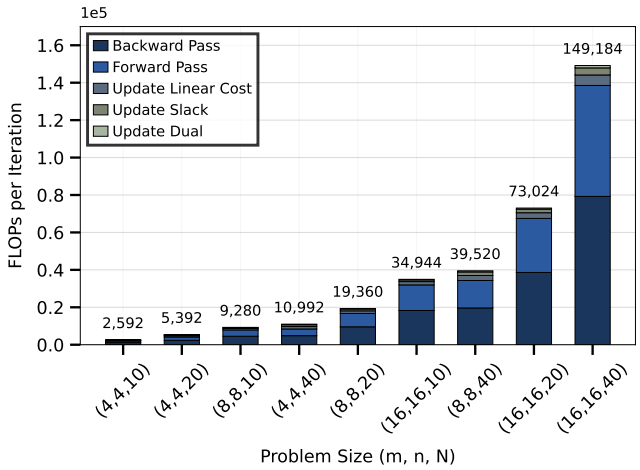

Fig. 4. Breakdown of FLOPs per iteration of various problem sizes

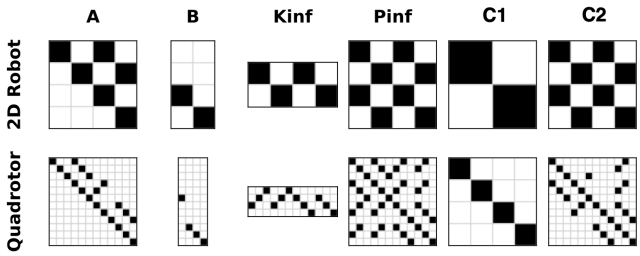

Fig. 5. Sparsity patterns of matrices that are used in TinyMPC computation. Black squares represent non-zero values.

Method of Multipliers (ADMM) [4]. The remaining minimization of a quadratic cost subject to linear dynamics constraints is the canonical Linear Quadratic Regulator (LQR) [17] which admits a closed-form affine feedback control solution of the form $u_k = -K_k x_k - d_k$. TinyMPC then goes further and precomputes and caches the approximate infinite-horizon LQR solution, K_{inf} and P_{inf}, as well as the terms $C_1 = (R + B^T P_{\text{inf}} B)^{-1}$ and $C_2 = (A - BK_{\text{inf}})^T$. This combination of tricks enables TinyMPC to leverage a compact cache online and eliminate the need for online matrix factorization, reducing computational complexity from cubic to quadratic, and enabling its deployment onto computationally constrained tiny robots for real-world dynamic robotic demonstrations. Psuedocode for the final algorithm is shown in Figure 2.

Figure 3 illustrates the computational flow of TinyMPC, where the orange path highlights the dominant floating-point multiplication operations. The inherent sequential data dependencies in this flow pose significant challenges for parallel computation. The problem scale is characterized by three parameters: the state dimension (n), control dimension (m), and time horizon (N). Figure 4 reveals that the computational bottleneck primarily occurs during the `forward_pass` and `backward_pass` stages, which involve intensive matrix-vector multiplications whose complexity scales quadratically with problem size. This profiling confirms that optimizing matrix-vector multiplication efficiency should be the central focus of our hardware architecture design.

Another computational characteristic is the consistent sparsity patterns inside the various matrices, as demonstrated in Figure 5. A detailed analysis of the formation of these sparsity patterns for the system dynamics is provided in Section IV-B. Building on prior work [24], [25], we leverage these structural properties for further optimizations in our hardware design.

C. Architecture Design Challenges and Solutions

We identify three key challenges in designing efficient MPC hardware accelerators and present our corresponding solutions:

- **Challenge 1: Scalability across problem sizes and platforms.** A fixed architecture with predetermined vector/matrix sizes cannot optimally adapt to varying control system scales under different FPGA resource constraints. **Solution**: We develop a parameterized architecture template coupled with a coarse-grained instruction set. The template supports flexible instantiation of function unit widths and depths to match both problem dimensions and platform resources. Our instruction set operates at matrix/vector granularity, enabling computation flows to remain portable across different hardware configurations.
- **Challenge 2: Sequential data dependencies.** The inherent sequential updates of state vectors (x, d, p) resist straightforward data-level parallelization.
  Solution: We exploit instruction-level parallelism through HLS code orchestration. Our framework automatically explores parallelization opportunities using the HLS compiler's built-in scheduler on generated code, eliminating manual architecture tuning.
- **Challenge 3: System-specific sparsity utilization.** System dynamics matrices $(\mathbf{A}, \mathbf{B})$ exhibit application-specific sparsity patterns that generic designs cannot efficiently exploit.
  Solution: Our compiler automatically generates HLS code with hard-coded sparsity patterns, enabling the elimination of redundant operations during synthesis while

TABLE I
INSTRUCTION SET OF TINYPROCESSOR

Type	Example	Description
Scalar	`fmul F0, F2, F3`	scalar multiplication
Vector	`vmax V0, V1, V2, vlen`	element-wise variable saturation
GeMV	`gemv M0, V0, V1, vlen`	matrix-vector multiplication
Branch	`bne R0, R1, target`	branch instruction

maintaining the original problem structure.

III. PARAMETERIZED AND PROGRAMMABLE VECTOR ARCHITECTURE

To begin to tackle these challenges, and in particular Challenge 1, we present a flexible processor architecture, illustrated in Figure 6, that offers configurable hardware and software components. On the software side, we provide a specialized instruction set for developing assembly-level implementations of MPC solver workflows. The hardware architecture employs a parameterized template design that supports scalable parallelism through configurable vector and matrix processing units. The coarse granularity of the designed instruction set allows different hardware configurations to share the same assembly description of the computation flow. This adaptable framework ensures optimal implementation across different target platforms with varying area constraints.

A. Instruction Set and Software

The proposed processor's instruction set, detailed in Table I, supports floating-point and integer scalar operations, floating-point vector operations, matrix-vector multiplications, and branch instructions. As demonstrated in Figure 7, through an assembly segment of the TinyMPC algorithm, this implementation efficiently encodes MPC computations while retaining the generality needed for other cached MPC workloads.

B. Parametrized Architecture Template

The core architectural template comprises three configurable function units: a scalar processing unit, a vector processing unit, and a dedicated GeMV (General Matrix-Vector multiply) unit, with their parameterized structure detailed in Figure 6. While the GeMV and vector units share computational primitives including multiply-accumulate operations, their physical separation enables future investigation of mixed-precision hardware implementations.

The proposed architecture template, coined as the Parametrized and programmable Vector Architecture (PVA), is instantiated using Vitis HLS to facilitate efficient exploration of the design space. The memory subsystem configuration is determined by two key parameters: vDepth controlling the vector buffer capacity and mDepth governing the matrix storage depth. To optimize memory bandwidth and computational throughput, the design employs a pre-loading strategy for vector and matrix operands. The vector unit achieves parallel processing of vWidth elements per cycle, while the GeMV operation's parallelism is determined by the multiply-accumulate tree parameters MacTreeWidth and nMacTree.

IV. APPLICATION-DRIVEN FUSED ARCHITECTURE

The proposed PVA design can handle a variety of MPC applications through software reprogramming. In real-world applications, the variables in an MPC problem can be divided into constant variables and time-varying variables. For example, in a trajectory tracing task of a quadrotor, the $\mathbf{A}$, $\mathbf{B}$ matrices and trajectory are fixed, while x is time-varying.

As such, to begin to address our other core challenges, we propose a fused design which offers key advantages for MPC scenarios where the dynamic characteristic matrices are fixed and computational workflows are static. First, it eliminates the resource overhead of general-purpose processing units by employing optimized dedicated hardware. Second, it removes the instruction scheduling overhead entirely through static, compile-time optimization of the fixed computation patterns.

A. Operation fusion

In programmable vector architectures, programs are typically compiled to binary code for execution on general-purpose processors. To optimize problem-specific MPC solvers, we leverage *partial evaluation* from compiler theory, generating specialized HLS C code tailored to the static parameters of the MPC problem. This specialized code is subsequently synthesized into optimized hardware circuits on the FPGA platform, eliminating runtime overhead for fixed computations. This idea is described as Figure 8, as we can see, the schedule in the fused design increases efficiency by breaking the original ordering. Furthermore, the code snippet shown in Figure 9 demonstrates such fused forward pass operations (corresponding to Step 1 in Figure 3), consolidating what would require 5 discrete instructions in the original non-fused architecture. This fusion is enabled by partial evaluation of the static computation graph structure.

B. Hardware Pruning by Constant Folding

The partial evaluation approach can be extended to enable hardware pruning by leveraging matrix sparsity patterns. For example, Equation 2 describes the dynamics of a 2D robot and illustrates how physical laws induce sparsity. For this 2D robot, directional motions are independent, yielding a block-diagonal structure with zeros isolating each coordinate.

$$\text{Discrete system: } \begin{cases} x_{k+1} = x_k + \Delta t\, v_{x,k} \\ y_{k+1} = y_k + \Delta t\, v_{y,k} \\ v_{x,k+1} = v_{x,k} + \Delta t\, a_x \\ v_{y,k+1} = v_{y,k} + \Delta t\, a_y \end{cases} \tag{2}$$

$$\Rightarrow \quad \mathbf{A} = \begin{bmatrix} 1 & 0 & \Delta t & 0 \\ 0 & 1 & 0 & \Delta t \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \quad \mathbf{B} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ \Delta t & 0 \\ 0 & \Delta t \end{bmatrix}$$

Similarly, linearizing a quadrotor about a hover results in sparse horizontal-vertical decoupling and diagonal rotational actuation as shown in the $\mathbf{A}, \mathbf{B}$ matrices in Figure 5. This sparsity is universal in physics-based robot models—and is

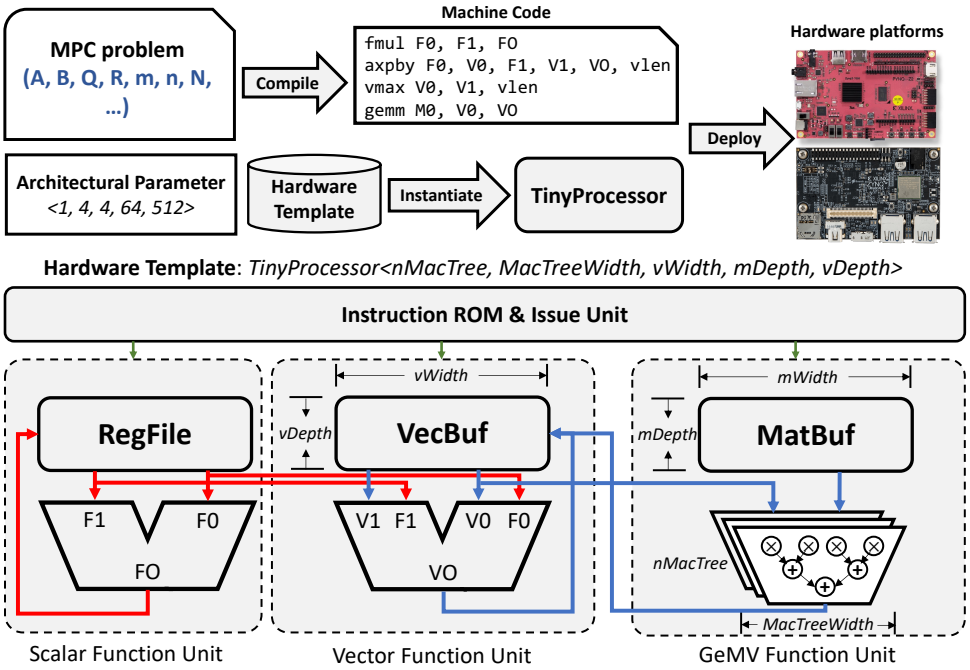

Fig. 6. The overall workflow to deploy an MPC problem onto our programmable vector architecture. In the figure, Red and blue lines denote the scalar and vector data paths, respectively.

structural, not a numerical artifact—enabling design-time optimizations to improve computational efficiency. For example, real-world actuators (thrust, torques) affect narrow subsets of states, leaving most matrix entries zero.

In our current implementation, this optimization is achieved through automatically generated HLS code that eliminates operations involving zero-valued operands, as demonstrated in Figure 10. This further address our challenges and leads to a performant, and custom, hardware generation.

V. Performance Evaluation

This section presents a thorough evaluation of our hardware solver, analyzing its computational speed, resource utilization, power consumption, and end-to-end performance. The proposed programmable vector architecture provides a scalable solution for control problems of varying scales. We further demonstrate performance improvements through problem-specific optimizations in the fused computation engine.

A. Experimental Setup

Our evaluation platform is the Avnet Ultra96 development board, equipped with a Zynq UltraScale+ MPSoC featuring a quad-core ARM Cortex-A53 application processor, dual-core Cortex-R5 real-time processors, and programmable logic fabric. For microcontroller (MCU) benchmarks, we generate executables using the `codegen` tool from the TinyMPC Python package [21], which are then executed on the processing system (PS). The hardware solver is implemented with Vitis HLS 2021.2 and deployed to the programmable logic (PL) section. Hardware specifications are provided in Table II.

```
li R1, 0
li R2, max_iter
main:
    # Set the arguments for forward pass
    li R8, N
    addi R8, -1, R8
    li R9, BASE_x
    li R10, BASE_u
    li R11, BASE_d
    jal RA, forward_pass
    ...
    jal RA, update_slack
    ...
    addi R1, 1, R1
    bne R1, R2, main
    halt

forward_pass:
    li R4, 0
    li R5, BASE_vtemp0
    li R6, BASE_vtemp1
forward_pass_loop:
    #  vTemp0 = Kinf * x
    gemv BASE_Kinf, R9, R5, nx, nx
    #  u = -Kinf * x - d
    axpby minus_one, R5, minus_one, R11, R10, nu
    #  vTemp0 = A * x
    gemv BASE_A, R9, R5, nx, nx
    #  vTemp1 = B * u
    gemv BASE_B, R10, R6, nx, nu
    #  x = A * x + B * u
    ...
    bne R4, R8, forward_pass_loop
    jr RA
```

Fig. 7. The assembly code implementation of TinyMPC.

B. Performance Evaluation of PVA

We first assess the PVA performance in solving MPC problems across different scales. The processor we used has the following architectural parameters: (vDepth=1024, vWidth=32, MacTreeWidth=8, nMacTree=4, mDepth=1024).

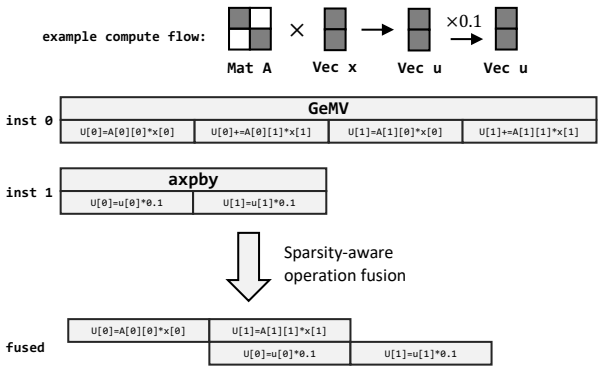

Fig. 8. A graphical comparison showing the differences between instruction-based execution and fused operations.

```
// --- Forward Pass ---
FWD_PASS_LOOP: for (int k = 0; k < 9; ++k) {
#pragma HLS PIPELINE
        U(k, 0) = -(Kinf_const[0][0] * X(k, 0) +
             Kinf_const[0][1] * X(k, 1) +
             Kinf_const[0][2] * X(k, 2) +
             Kinf_const[0][3] * X(k, 3)) - D(k, 0);
        U(k, 1) = -(Kinf_const[1][0] * X(k, 0) +
             Kinf_const[1][1] * X(k, 1) +
             Kinf_const[1][2] * X(k, 2) +
             Kinf_const[1][3] * X(k, 3)) - D(k, 1);

        X(k+1, 0) = (A_const[0][0] * X(k, 0) +
             A_const[0][1] * X(k, 1) +
             A_const[0][2] * X(k, 2) +
             A_const[0][3] * X(k, 3)) +
             (B_const[0][0] * U(k, 0) +
             B_const[0][1] * U(k, 1));
        X(k+1, 1) = (A_const[1][0] * X(k, 0) +
             A_const[1][1] * X(k, 1) +
             A_const[1][2] * X(k, 2) +
             A_const[1][3] * X(k, 3)) +
             (B_const[1][0] * U(k, 0) +
             B_const[1][1] * U(k, 1));

        /* ... */
    }
```

Fig. 9. Optimized forward pass implementation with fused operations.

FPGA resource costs of implementing this processor are shown in Table III. The evaluation employs three test series that systematically vary one problem dimension (m, n, or N) while fixing the other two parameters.

With a fixed maximum iteration count of 10,000, we measure the computation time per iteration for each configuration. The test data is randomly generated. As shown in Figure 11, the PVA demonstrates consistent speedups over the software baseline across all test cases. The measured performance improvements range from 1.46× (minimum) to 2.70× (maximum), with an average speedup of 2.61×.

C. Performance Evaluation of the Fusion Engine

To evaluate our application-specific design methodology (Section IV), we conduct ablation studies on two trajectory-tracking applications: *2D robot* and *quadrotor* with planning horizons of 10 and 30. The breakdown of total computation

```
// --- Forward Pass ---
    FWD_PASS_LOOP: for (int k = 0; k < 9; ++k) {
#pragma HLS PIPELINE
        U(k, 0) = -(2.110300282111507f * X(k, 0)
             + 2.417446239653999f * X(k, 2)) -
             D(k, 0);
        U(k, 1) = -(2.110300282111507f * X(k, 1)
             + 2.417446239653999f * X(k, 3)) -
             D(k, 1);

        X(k+1, 0) = (1.0f * X(k, 0) + 0.1f * X(k,
             2));
        X(k+1, 1) = (1.0f * X(k, 1) + 0.1f * X(k,
             3));
        X(k+1, 2) = (1.0f * X(k, 2)) + (0.1f *
             U(k, 0));
        X(k+1, 3) = (1.0f * X(k, 3)) + (0.1f *
             U(k, 1));
    }
```

Fig. 10. Optimized forward pass implementation with sparsity pattern-aware constant folding.

TABLE II
ULTRA96 KEY HARDWARE SPECIFICATIONS

PL				PS	
LUTs	**FFs**	**DSPs**	**BRAMs**	**Processor**	**Memory**
70,560	141,120	360	216	4× Cortex-A53 @1.5GHz	2GB DDR4

is shown in Figure 12, showing that the pruning technique mentioned in IV-B significantly reduces the amount of FLOPs.

The resource costs of each fused engine are listed in Table III, which shows that the fused engine not only provides control rate improvements but also significantly reduces the hardware resource usage. Key findings demonstrate progressive performance gains: The baseline PVA architecture matches software solution performance with a 0.97× speedup. Building on this foundation, operation fusion achieves a 3.05× speedup through compile-time operation fusion as detailed in Section IV. Further enhancing performance, sparsity exploitation yields an additional 1.79× speedup by leveraging physics-induced structural sparsity patterns shown in Fig. 5.

The combined optimizations deliver 5.46× total speedup versus software baseline. In the Quadrotor-N30 task, the fused engine delivers 9.73× speedup. This demonstrates: (1) specialized hardware eliminates scheduling overhead, (2) static computation patterns enable deep optimization, and (3) physical sparsity (Fig. 5) substantially reduces complexity (Fig. 12).

D. Power & Energy Efficiency

We measured both static and dynamic power consumption across all solver systems. The power is measured by the on-board PMBus sensors on the Ultra96. Static power was measured while the solver remained idle, whereas dynamic power was calculated as the difference between active solver execution and idle power states. Our results demonstrate that while the static power consumption of MCU and PVA architectures is comparable, the PVA exhibits a 53.27 % reduction in dynamic power compared to the MCU implementation.

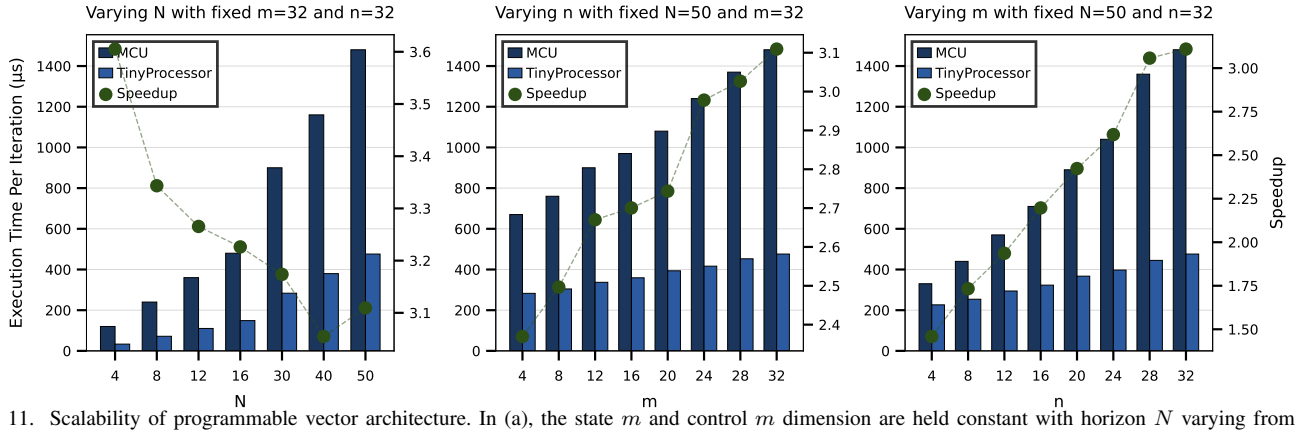

Fig. 11. Scalability of programmable vector architecture. In (a), the state m and control m dimension are held constant with horizon N varying from 4 to 50. In (b), the N and m are held constant with m varying from 4 to 32. In (c), the N and n are held constant with n varying from 4 to 32.

TABLE III
RESOURCE UTILIZATION COMPARISON

Arch.	LUTs		FFs		DSPs		BRAMs	
	Used	%	Used	%	Used	%	Used	%
PVA	44.7k	43.4	38.7k	27.4	232	64.4	67	31.0
2D-N10	10.6k	15.0	12.9k	9.1	40	11.1	9	4.2
2D-N30	10.8k	15.4	13.0k	9.2	40	11.1	9	4.2
Q-N10	25.7k	36.4	25.2k	17.9	104	28.9	9	4.2
Q-N30	26.0k	36.9	25.3k	17.9	104	28.9	9	4.2

TABLE IV
POWER CONSUMPTION AND ENERGY EFFICIENCY COMPARISON BETWEEN ARCHITECTURES

	Static (mW)	Dynamic (mW)
MCU	597	122
PVA	593	57
2D-N10	250	30
2D-N30	250	59
Q-N10	416	28
Q-N30	416	56

The fused architecture demonstrates significant improvements in both static and dynamic power consumption compared to conventional MCU and PVA solutions, while simultaneously achieving higher computational throughput. As demonstrated in the Quadrotor-N30 benchmark, the fused implementation achieves a 54.10 % power reduction while maintaining a 9.73× speedup over the MCU solution.

E. End-to-end Performance Evaluation

To assess the performance of our hardware implementation, we conducted comparative trajectory tracking experiments using both software and hardware-based MPC solvers. The baseline software implementation employs a prediction horizon of $N = 10$, while our hardware implementations feature three configurations with horizons of $N = 10$, $N = 20$, and $N = 30$. The control frequency is calculated as $Freq = 0.5 \times \frac{1}{t_{solution}}$, where the scaling factor of 0.5 ensures adequate time margin for both computation and actuation. The software

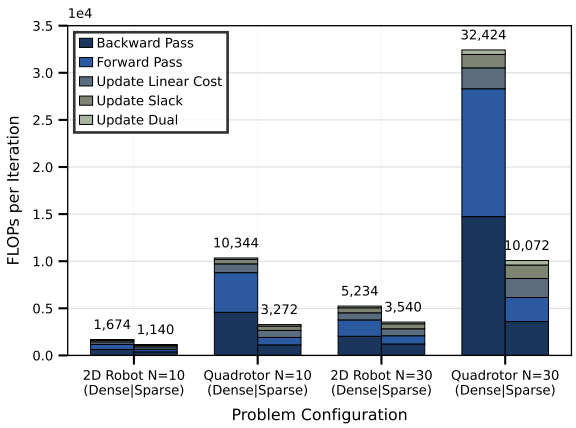

Fig. 12. Breakdown of total FLOPs in dense computation and sparse-aware computation.

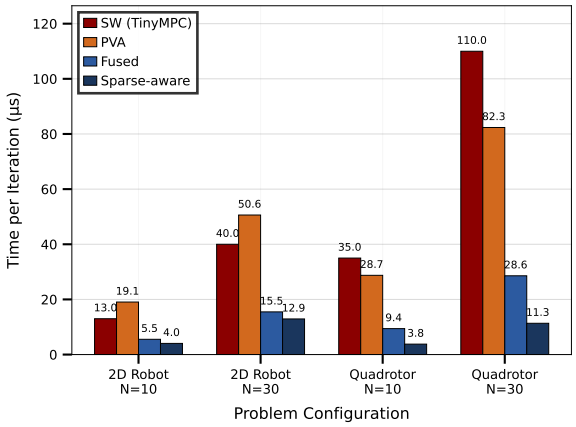

Fig. 13. Comparison of solution time per iteration (the lower the better) under different hardware configurations.

solver achieves a control frequency of 142 Hz, while the hardware implementations attain frequencies of 205 Hz, 174 Hz, and 137 Hz for horizons 10, 20 and 30, respectively.

In our experiments, we simulated the quadrotor's spiral trajectory while incorporating random noise to better approximate real-world conditions. As shown in Figure 14, the results demonstrate that our hardware solver successfully enables MPC operation with both extended planning horizons and im-

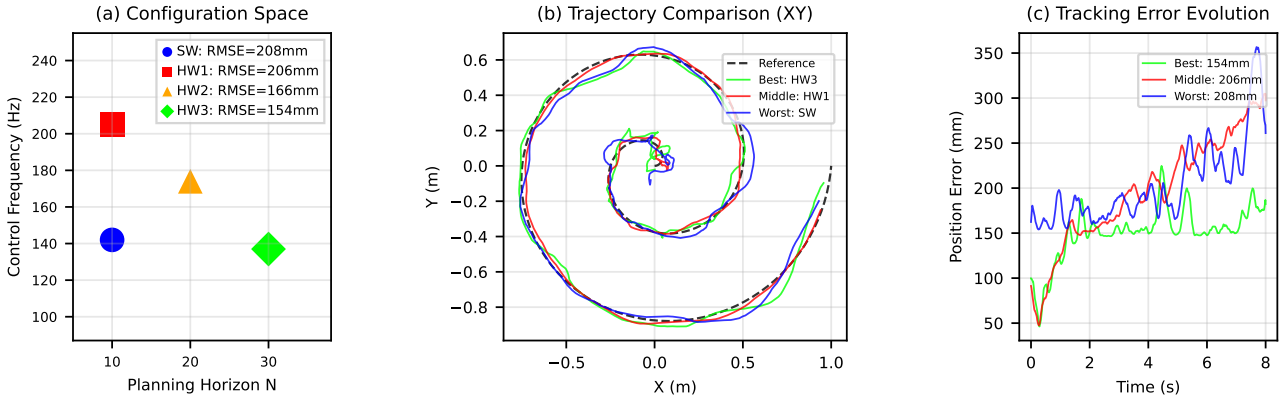

Fig. 14. An end-to-end performance comparison in spiral trajectory tracing task between hardware and software solver.

proved control frequencies – two critical factors for enhanced control performance. This improvement is reflected in the reduction of tracking error from 208 mm to 154 mm. With the aid of a hardware accelerator, we can support longer planning horizons and higher control frequencies, which ultimately improves the control performance.

VI. Conclusion and Future Work

In this work, we present two key design approaches integrated into a comprehensive framework for implementing embedded MPC solvers on FPGA platforms: (1) a programmable vector architecture offering runtime flexibility, and (2) an application-driven fused engine optimized for specific models. The programmable vector architecture provides rapid adaptability to new problem instances through its reconfigurable datapath, while the fused engine achieves superior performance through model-specific hardware customization.

Our evaluation demonstrates substantial improvements over conventional microcontroller implementations. The PVA achieves a $2.61\times$ speedup with a $53.27\,\%$ power reduction, while the fused engine delivers more aggressive optimization with a $9.72\times$ acceleration and a $54.10\,\%$ power savings for quadrotor control applications. Furthermore, through an end-to-end trajectory tracking case study, we empirically validate that our hardware-accelerated solver enhances closed-loop control performance.

In future work, we hope to expand our framework to encompass and evaluate additional (cached) edge MPC algorithms (e.g., [3]) and to deploy our designs onto physical robots.

References

[1] D. Arnström, A. Bemporad, and D. Axehill, "A Dual Active-Set Solver for Embedded Quadratic Programming Using Recursive LDL^T Updates," *IEEE Transactions on Automatic Control*, vol. 67, no. 8, pp. 4362–4369, Aug. 2022.

[2] J. T. Betts, *Practical Methods for Optimal Control Using Nonlinear Programming*, ser. Advances in Design and Control. Society for Industrial and Applied Mathematics (SIAM), 2001, vol. 3.

[3] A. L. Bishop, J. Z. Zhang, S. Gurumurthy, K. Tracy, and Z. Manchester, "ReLU-QP: A GPU-accelerated quadratic programming solver for model-predictive control," in *2024 IEEE International Conference on Robotics and Automation (ICRA)*. Yokohama, JP: IEEE, 2024, pp. 13 285–13 292.

[4] S. Boyd, N. Parikh, E. Chu, B. Peleato, J. Eckstein *et al.*, "Distributed optimization and statistical learning via the alternating direction method of multipliers," *Foundations and Trends® in Machine learning*, vol. 3, no. 1, pp. 1–122, 2011.

[5] F. Dong, X. Li, K. You, and S. Song, "Standoff tracking using dnn-based mpc with implementation on fpga," *IEEE Transactions on Control Systems Technology*, vol. 31, no. 5, pp. 1998–2010, 2023.

[6] U. Eren, A. Prach, B. B. Koçer, S. V. Raković, E. Kayacan, and B. Açıkmeşe, "Model predictive control in aerospace systems: Current state and opportunities," *Journal of Guidance, Control, and Dynamics*, vol. 40, no. 7, pp. 1541–1566, 2017.

[7] C. E. Garcia, D. M. Prett, and M. Morari, "Model predictive control: Theory and practice—a survey," *Automatica*, vol. 25, no. 3, pp. 335–348, 1989.

[8] S. Gerkšič and B. Pregelj, "Finite-word-length FPGA implementation of model predictive control for ITER resistive wall mode control," *Fusion Engineering and Design*, vol. 169, p. 112480, Aug. 2021.

[9] E. N. Hartley, J. L. Jerez, A. Suardi, J. M. Maciejowski, E. C. Kerrigan, and G. A. Constantinides, "Predictive control using an FPGA with application to aircraft control," *IEEE Transactions on Control Systems Technology*, vol. 22, no. 3, pp. 1006–1017, 2013.

[10] M. He and K. V. Ling, "Model Predictive Control on a Chip," in *2005 International Conference on Control and Automation (ICCA)*. Budapest, Hungary: IEEE, 2005, pp. 528–532.

[11] M. Jeong, M. Schoen, and J. 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)*, Jeju Island, Korea, Republic of Korea, May 2023, pp. 1704–1711.

[12] J. L. Jerez, G. A. Constantinides, and E. C. Kerrigan, "FPGA Implementation of an Interior Point Solver for Linear Model Predictive Control," in *2010 International Conference on Field-Programmable Technology (FPT)*. Beijing, CHina: IEEE, 2010, pp. 316–319.

[13] J. L. Jerez, G. A. Constantinides, and E. 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*. Monterey, CA: ACM, 2011, pp. 209–218.

[14] J. L. Jerez, P. J. Goulart, S. Richter, G. A. Constantinides, E. C. Kerrigan, and M. Morari, "Embedded Online Optimization for Model Predictive Control at Megahertz Rates," *IEEE Transactions on Automatic Control*, vol. 59, no. 12, pp. 3238–3251, 2014.

[15] G. Knagge, A. Wills, A. Mills, and B. Ninness, "ASIC and FPGA implementation strategies for Model Predictive Control," in *Proceedings of the 2009 European Control Conference (ECC)*. Budapest, Hungary: IEEE, 2009, pp. 144–149.

[16] S. Kuindersma, "Taskable agility: Making useful dynamic behavior easier to create," Princeton Robotics Seminar, 4 2023.

[17] F. L. Lewis, D. Vrabie, and V. L. Syrmos, *Optimal control*. John Wiley & Sons, 2012.

[18] Y. Li, S. E. Li, X. Jia, S. Zeng, and Y. Wang, "FPGA accelerated model predictive control for autonomous driving," *Journal of Intelligent and Connected Vehicles*, vol. 5, no. 2, pp. 63–71, Jan. 2022.

[19] J. Liu, H. Peyrl, A. Burg, and G. 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)*. Montreal, QC, Canada: IEEE, 2014, pp. 1–8.

[20] S. Lucia, D. Navarro, B. Karg, H. Sarnago, and Ó. Lucía, "Deep Learning-Based Model Predictive Control for Resonant Power Converters," *IEEE Transactions on Industrial Informatics*, vol. 17, no. 1, pp. 409–420, Jan. 2021.

[21] I. Mahajan, K. Nguyen, S. Schoedel, E. Nedumaran, M. Mata, B. Plancher, and Z. Manchester, "Code generation and conic constraints for model-predictive control on microcontrollers with conic-tinympc," 2025.

[22] J. Mattingley and S. Boyd, "CVXGEN: A code generator for embedded convex optimization," *Optimization and Engineering*, vol. 13, pp. 1–27, 2012.

[23] I. McInerney, G. A. Constantinides, and E. C. Kerrigan, "A Survey of the Implementation of Linear Model Predictive Control on FPGAs," *IFAC-PapersOnLine*, vol. 51, no. 20, pp. 381–387, 2018.

[24] S. M. Neuman, R. Ghosal, T. Bourgeat, B. Plancher, and V. J. Reddi, "Roboshape: Using topology patterns to scalably and flexibly deploy accelerators across robots," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*. Orlando, FL, USA: ACM, 2023, pp. 1–13.

[25] S. M. Neuman, B. Plancher, T. Bourgeat, T. Tambe, S. Devadas, and V. J. Reddi, "Robomorphic computing: a design methodology for domain-specific accelerators parameterized by robot morphology," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, Virtual USA, 2021, pp. 674–686.

[26] K. Nguyen, S. Schoedel, A. Alavill, B. Plancher, and Z. Manchester, "TinyMPC: Model-Predictive Control on Resource-Constrained Microcontrollers," in *IEEE International Conference on Robotics and Automation (ICRA)*. Yokohama, Japan: IEEE, May 2024.

[27] H. Peyrl, A. Zanarini, T. Besselmann, J. Liu, and M. A. Boéchat, "Parallel implementations of the fast gradient method for high-speed MPC," *Control Engineering Practice*, vol. 33, pp. 22–34, 2014.

[28] S. J. Qin and T. A. Badgwell, "A survey of industrial model predictive control technology," *Control engineering practice*, vol. 11, no. 7, pp. 733–764, 2003.

[29] Y. Shi and K. Zhang, "Advanced model predictive control framework for autonomous intelligent mechatronic systems: A tutorial overview and perspectives," *Annual Reviews in Control*, vol. 52, pp. 170–196, 2021.

[30] B. Stellato, G. Banjac, P. Goulart, A. Bemporad, and S. Boyd, "OSQP: an operator splitting solver for quadratic programs," *Mathematical Programming Computation*, vol. 12, no. 4, pp. 637–672, 2020.

[31] Q. Sui, B. Du, Y. Zuo, and W. Martinez, "Exploring the Potential of FPGA in High-Frequency Switching DC-DC Boost Converters Using Model Predictive Control," in *2025 IEEE Applied Power Electronics Conference and Exposition (APEC)*. Atlanta, GA, USA: IEEE, Mar. 2025, pp. 2752–2756.

[32] S. Vazquez, J. I. Leon, L. G. Franquelo, J. Rodriguez, H. A. Young, A. Marquez, and P. Zanchetta, "Model predictive control: A review of its applications in power electronics," *IEEE industrial electronics magazine*, vol. 8, no. 1, pp. 16–31, 2014.

[33] P. M. Wensing, M. Posa, Y. Hu, A. Escande, N. Mansard, and A. D. Prete, "Optimization-based control for dynamic legged robots," *IEEE Transactions on Robotics*, vol. 40, pp. 43–63, 2024.

[34] A. G. Wills, G. Knagge, and B. Ninness, "Fast Linear Model Predictive Control Via Custom Integrated Circuit Architecture," *IEEE Transactions on Control Systems Technology*, vol. 20, no. 1, pp. 59–71, 2012.

[35] Y. Xu, D. Li, Y. Xi, J. Lan, and T. Jiang, "An Improved Predictive Controller on the FPGA by Hardware Matrix Inversion," *IEEE Transactions on Industrial Electronics*, vol. 65, no. 9, pp. 7395–7405, 2018.