An Embedded Scalable Linear Model Predictive Hardware-based Controller using ADMM

Pei Zhang, Joseph Zambreno and Phillip H. Jones
Electrical and Computer Engineering
Iowa State University, Ames, Iowa, USA
{peizhang, zambreno and phjones@iastate.edu}

***Abstract*—Model predictive control (MPC) is a popular advanced model-based control algorithm for controlling systems that must respect a set of system constraints (e.g. actuator force limitations). However, the computing requirements of MPC limits the suitability of deploying its software implementation into embedded controllers requiring high update rates. This paper presents a scalable embedded MPC controller implemented on a field-programmable gate array (FPGA) coupled with an on-chip ARM processor. Our architecture implements an Alternating Direction Method of Multipliers (ADMM) approach for computing MPC controller commands. All computations are performed using floating-point arithmetic. We introduce a software/hardware (SW/HW) co-design methodology, for which the ARM software can configure on-chip Block RAM to allow users to 1) configure the MPC controller for a wide range of plants, and 2) update at runtime the desired trajectory to track. Our hardware architecture has the flexibility to compromise between the amount of hardware resources used (regarding Block RAMs and DSPs) and the controller computing speed. For example, this flexibility gives the ability to control plants modeled by a large number of decision variables (i.e. a plant model using many Block RAMs) with a small number of computing resources (i.e. DSPs) at the cost of increased computing time. The hardware controller is verified using a Plant-on-Chip (PoC), which is configured to emulate a mass-spring system in real-time. A major driving goal of this work is to architect an SW/HW platform that brings FPGAs a step closer to being widely adopted by advanced control algorithm designers for deploying their algorithms into embedded systems.**

***Index Terms*—MPC, FPGA, ADMM, SW/HW co-design, co-processor, control**

I. Introduction

Model predictive control (MPC) is a popular advanced control algorithm for controlling systems that must respect a set of system constraints (e.g. actuator force limitations). MPC has found its way into a wide range of applications, such as industrial chemical plants [1], power converters [2], traffic networks [3], and unmanned aerial vehicles(UAVs) [4]. However, for two decades after its introduction in the late 1970's by J. Richalet [5], this advanced control technique gained little traction outside of systems requiring update rates on the order of seconds to minutes [6]. A primary reason for its lack of adoption as compared to other control strategies such as proportional-integral-derivative (PID) and optimal linear quadratic control was due to its intense computing demands.

Methods for using limited computing resources to increase MPC update rates has become a central problem for deploying MPC controllers into embedded systems for controlling increasingly complex systems at KHz and closing in on MHz update rates. In this paper we use the parallelizable operator splitting method, also referred to as alternating directions method of multipliers (ADMM), to solve the linear-quadratic MPC problem. Apart from its parallelizability, the algorithm is division-free [7]. We build the design for a Zynq-7020 Field Programmable Gate Array (FPGA) device to exploit its potential computation parallelism. Our proposed design targets control algorithm and embedded software system developers that wish to make use of the computing capabilities of FPGAs to accelerate MPC, but may have little knowledge of FPGA hardware design.

Contributions. The primary contribution of this work is a software/hardware (SW/HW) co-design that allows: 1) configuring an MPC controller for a wide range of plants, 2) updating at run-time the desired trajectory to track, 3) the flexibility to trade off hardware resources for computing speed, and 4) easing controller deployment by introducing an SW/HW co-design to decouple hardware details from control and embedded software engineers.

Organization. The remainder of this paper is organized as follows. Section II reviews works related to MPC computation. Section III then presents a brief summary of MPC basics using state space modeling and the concept of ADMM. Section IV gives the hardware architecture and analyzes its bottlenecks. Section V evaluates the hardware resource usage, maximum clock frequency, and provides experimental controller results. Section VI concludes the paper.

II. Related Work

In this section techniques for computing MPC as a Quadratic Programming problem are discussed. A summary is then given of state-of-the-art FPGA-based hardware acceleration implementations for these techniques. We then position our approach within those works.

Quadratic Programming (QP) solutions. MPC can be posed as a Quadratic Programming problem in which a quadratic cost function is optimized subject to a set of linear equality and inequality constraints. As compared to computing Linear Quadratic Regulator (LQR) commands, which can be posed as a QP problem that is subject to only linear equalities, computing MPC commands require vastly more computing resources. There are two key reasons for this. First, when only equality constraints are considered there is an analytical

978-1-5090-4825-0/17/$31.00

solution, while iterative methods are required once inequality constraints are introduced. Second, LQR only uses the current system state and sensor inputs to compute its next actuator command, while MPC additionally uses predictions of system state and sensor inputs over a specified number of time steps into the future (i.e. prediction horizon) [8].

QP problems can be solved reliably via various iterative methods. Three common methods are the: 1) Interior-Point Method (IPM) [9, Chapter 4.3.2], 2) the Active Set Method (ASM) [9, Chapter 4.3.3], and 3) Alternating Directions Method of Multipliers (ADMM) [7], [10]. For IPM, each inequality constraint is transformed into a sequence of equality constrained problems, and solved using Newton's method. ASM selects a subset of all specified inequalities based on which inequalities are currently "active", meaning they will affect the optimization result at the current stage of the computation. ADMM (also called operator splitting) allows large QP problems to be broken into a set of smaller pieces, thus allowing for more opportunities for parallelism. A detailed description of IPM and ASM can be found in [11], [10] gives a comprehensive introduction to ADMM, and a good reference for solutions to the more general problem of convex optimization is [12].

In most cases, IPM requires fewer iterations than ASM to converge. However, each iteration of IPM is more computationally expensive because it solves a linear system involving all the variables of the problem, whereas ASM solves linear systems involving a subset of all the variables (i.e. variables associated with a subset of active constraints). ADMM converges slower than IPM and ASM to achieve the same accuracy, while each iteration is easier to compute.

FPGA-based QP solutions. Several works have investigated accelerating QP solutions for the purpose of MPC [13]–[19]. Hardware acceleration of MPC using IPM was performed by [13]–[16]. For these works accelerating the linear equation solver required for IPM was the focus. In [13], the MINRES algorithm was used to exploit potential parallelism within the linear equation solver. In [14] and [16], acceleration of a Conjugate Gradient Method based linear solver was conducted. In [15], the linear equation solver used a Cholesky decomposition approach that enabled implementing a predictor-corrector that reduced the number of solver iterations. An accelerator using the ASM approach was implemented in [17]. Additionally, [17] examined the trade-offs between using an ASM verses an IPM approach. They concluded that ASM gives lower computing complexity and converges faster when the number of decision variables and constraints are small. Otherwise, IPM is a better choice when considering scalability.

ADMM's inherent parallelizability makes it a natural fit for hardware acceleration. Two works that have developed ADMM acceleration engines are [18] and [19]. In [18], a highly parallel architecture was presented, and the tradeoff between accuracy and computing resources when using custom fixed-point number representation within the engine's core was a major focus. The high-level architecture of this work's computing core is the most similar to our work. In [19], the use of a sparse QP formulation under polytopic constraints was the primary contribution.

For our ADMM-based MPC acceleration engine, we have focused on a SW/HW co-design that is flexible and eases its use and deployment into a system. In terms of flexibility, our architecture allows scaling in such a way that computing speed can be traded off for hardware resources, enabling relatively large controllers to be deployed when only a small number of on-chip resources can be allocated to the engine. Regarding ease of use and deployment, we have tightly integrated our MPC engine with an on-chip ARM processor and we implement standard 32-bit floating point computations. Software running on the ARM processor makes updating the MPC engine with a new controller convenient, and our hardware architecture has implemented software settable ADMM tuning features as well.

III. Background

This section gives a brief overview of three topics important for understanding the problem being addressed: state space models, model predictive optimal control, and the splitting method.

A. State Space Model

In our paper, MPC is based on a state space model of a physical system. A discrete state-space model defines what state a system will be in one-time step into the future, based on the current state of the system and current input acting upon it. A generic linearized discrete state-space system model consists of matrices A, B, C, and D[1] and is formulated as follows:

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

$$y_k = Cx_k + Du_k \tag{2}$$

Where:

- x_k represents the state of the system at time k
- u_k represents the input acting on the system at time k
- y_k represents outputs of the system at time k
- A is a matrix that defines the internal dynamics of the system
- B is a matrix that defines how the input acting upon the system impact its state
- C is a matrix that transforms states of the system into outputs (y_k)

Equation (1) is referred to as the state update equation. With respect to a closed loop control system, matrix A represents the dynamics of the plant being controlled, matrix B represents how actuator commands (i.e. u_k) impact the plant, and the matrix C could be viewed as a mapping of the current state to the output obtained from sensors (i.e. y_k).

Also the width (i.e. number of columns) of each matrix or length of each vector found in Equation (1) and (2) can be viewed as follows:

[1]it is common to omit the matrix D, as inputs typically do not directly impact output

- M: the number of system/plant inputs/actuators.
- N: the number of system/plant states.
- P: the number of system outputs (i.e. sensors).
- $A \in \mathbb{R}^{N\times N}$; $B \in \mathbb{R}^{N\times M}$; $C \in \mathbb{R}^{P\times N}$.
- $x_k \in \mathbb{R}^N$; $y_k \in \mathbb{R}^P$; $u_k \in \mathbb{R}^M$.

B. Model Predictive Optimal Control

MPC uses Equation (1) and (2) to predict the behavior of the system from current time over a future prediction horizon, thus the input (u_k), input-change rate/step (Δu_k) and state (x_k) are augmented to cover future predictions, as in Formula (3).

$$U_k = \begin{bmatrix} u_k \\ u_{k+1} \\ \vdots \\ u_{k+H_u} \end{bmatrix}, \quad \Delta U_k = \begin{bmatrix} \Delta u_k \\ \Delta u_{k+1} \\ \vdots \\ \Delta u_{k+H_{u-1}} \end{bmatrix}, \quad X_k = \begin{bmatrix} x_k \\ x_{k+1} \\ \vdots \\ x_{k+H_p} \end{bmatrix} \tag{3}$$

Where:

- H_u: changeable future input horizon. We assume input u_k will be constant after H_u time steps.
- H_p: prediction horizon. Normally, $H_p \geq H_u$.
- $U_k \in \mathbb{R}^{M(H_u+1)}$, $\Delta U_k \in \mathbb{R}^{MH_u}$, $X_k \in \mathbb{R}^{N(H_p+1)}$.

1) MPC Cost Function: An MPC controller specifies a vector combination of all controlled variables(X_k) and constraint variables(ΔU_k, U_k), which minimizes the quadratic objective cost function given by Equation (4), subject to linear constraints on the variables x_i, u_i and Δu_i. q_i, p_i and s_i are positive cost constant applied to x_i, u_i and Δu_i.

$$\begin{aligned} \mathbb{C}(k) = \frac{1}{2}\Big(\sum_{i=k}^{k+H_p} (x_i^T q_i x_i - 2 r_i^T q_i x_i) + \sum_{i=k}^{k+H_u} u_i^T p_i u_i \\ + \sum_{i=k}^{k+H_{u-1}} \Delta u_i^T s_i \Delta u_i \Big) + Const \end{aligned} \tag{4}$$

Where r_k is a proposed trajectory at time k. Since the optimal solution is independent of $Const$, we omit this constant term and rewrite the cost function into more condense format:

$$\mathbb{C}(k) = \frac{1}{2} \begin{bmatrix} X_k \\ U_k \\ \Delta U_k \end{bmatrix}^T \begin{bmatrix} Q & & \\ & P & \\ & & S \end{bmatrix} \begin{bmatrix} X_k \\ U_k \\ \Delta U_k \end{bmatrix} - R_k^T Q X_k \tag{5}$$

Where R_k is the vector augmented form of r_k, and has the same format and size as X_k. The Q matrix is shown in (14) and the same format applies to the P and S matrix.

2) Constraints: The equality constraints are associated with the system time-step update, which is written as Equation (11). It contains the augmented form of Equation (1) and (2) and the equality relationship between U_k and ΔU_k. The inequalities are the restrictions on U_k, ΔU_k and X_k. Normally, the constraints to the state and input are called the box constraint, which sets a low boundary and a high boundary like an enclosed box that circumscribes the scope of these variables.

C. Splitting Method

One technique for partitioning variables in ADMM is writing the convex QP problem into consensus form [7]:

$$\begin{aligned} &minimize: \ \mathbb{1}_{\mathcal{D}}(\chi) + \phi(\chi) + \mathbb{1}_{\mathcal{C}}(\zeta) \\ &subject\ to: \ \chi = \zeta \end{aligned} \tag{6}$$

Where $\mathbb{1}_{\mathcal{D}}(\chi)$ is the indicator function: $\mathbb{1}_{\mathcal{D}} : \chi \to \{0, +\infty\}$. $\mathcal{D}$ is the affine set associated with system update in Equation (11). When χ fails to meet the equality condition, $\mathbb{1}_{\mathcal{D}}(\chi)$ goes infinity. So does $\mathbb{1}_{\mathcal{C}}(\zeta)$, where $\mathcal{C}$ is the variable constraints that the system must comply.

We use function $f + g$ to replace the objective function in QP problem (6). f and g are shown in in Equation (7). Starting from arbitrary points ζ^0 and υ^0, the operator splitting method solves the QP problem by carrying out three steps: (8) to (10) repeatedly. Each can be solved separately without division.

$$\begin{aligned} g(\chi) &= \mathbb{1}_{\mathcal{D}}(\chi) + \phi(\chi) \\ f(\zeta) &= \mathbb{1}_{\mathcal{C}}(\zeta) \end{aligned} \tag{7}$$

$$\chi^{i+1} := prox_{g,\rho}(\zeta^i + \upsilon^i) \tag{8}$$

$$\zeta^{i+1} := prox_{f,\rho}(\chi^{i+1} + \upsilon^i) \tag{9}$$

$$\upsilon^{i+1} := \upsilon^i + \rho(\chi^{i+1} - \zeta^{i+1}) \tag{10}$$

Here, i is the iteration counter, $prox_{f,\rho}(\chi)$ is the proximal mapping (or proximal operator) of a convex function f:

$$prox_{f,\rho}(\chi) = arg\ \min_u (f(u) + \frac{\rho}{2}\|\chi - u\|_2^2)$$

$\rho > 0$ is the dual update step length.

Computing χ^{i+1} in (8) is equivalent to solving χ^{i+1} in (11), which is the standard QP problem with equality constraints.

$$\begin{aligned} &minimize: \ \frac{1}{2}(\chi^{i+1})^T E \chi^{i+1} + l^T \chi^{i+1} \\ &subject\ to: \ G\chi^{i+1} = h \end{aligned} \tag{11}$$

The associated matrices/vectors are shown in Equation (12) to (14):

$$G = \begin{bmatrix} I & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ A & -I & \dots & \dots & \dots & \dots & B & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ \dots & A & -I & \dots & \dots & \dots & \dots & B & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & B & \dots & \dots & \dots & \dots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \dots & \dots & \dots & \dots & A & -I & \dots & \dots & \dots & \dots & \dots & B & \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots & I & -I & \dots & \dots & \dots & \dots & I & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots & I & -I & \dots & \dots & \dots & \dots & I & \dots & \dots & \dots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & I & -I & \dots & \dots & \dots & I \end{bmatrix} \tag{12}$$

$$\chi^{i+1} = \begin{bmatrix} x_k \\ x_{k+1} \\ x_{k+2} \\ \vdots \\ x_{k+H_p} \\ u_k \\ u_{k+1} \\ \vdots \\ u_{k+H_u} \\ \Delta u_k \\ \Delta u_{k+1} \\ \vdots \\ \Delta u_{k+H_u-1} \end{bmatrix}, \quad h = \begin{bmatrix} x_k \\ 0 \\ \vdots \\ 0 \end{bmatrix} \text{ and } \quad l = \begin{bmatrix} Q * R_k \\ \mathbf{0}_{(2H_u+1)M} \end{bmatrix} - \rho \begin{bmatrix} \zeta_0^i + \upsilon_0^i \\ \zeta_1^i + \upsilon_1^i \\ \vdots \\ \zeta_{S2-1}^i + \upsilon_{S2-1}^i \end{bmatrix} \tag{13}$$

$$E = \begin{bmatrix} Q+\rho I & & \\ & P+\rho I & \\ & & S+\rho I \end{bmatrix} \quad \text{and} \quad Q = \begin{bmatrix} q_0 & & \\ & q_1 & \\ & & \ddots \\ & & & q_{H_p} \end{bmatrix} \tag{14}$$

Let $S_1 = (H_p+1)N + H_u M$, and the number of optimization variables $S_2 = (H_p+1)N + (2H_u+1)M$, then $G \in \mathbb{R}^{S_1 \times S_2}$, $E \in \mathbb{S}_+^{S_2 \times S_2}$.

The common method to solve Equation (11) is to establish the KKT condition. The KKT condition is shown below:

$$\begin{bmatrix} E & G^T \\ G & \mathbf{0} \end{bmatrix} \begin{bmatrix} \chi^{i+1} \\ \lambda \end{bmatrix} = \begin{bmatrix} -l \\ h \end{bmatrix} \tag{15}$$

Where λ is called the dual variable vector. E is the diagonal combination of cost constant matrix Q, P and S as is shown in Equation (14). The KKT matrix is proved to be invertible [7].

We solve Equation (15) to obtain χ^{i+1}, which gives:

$$\begin{bmatrix} \chi^{i+1} \\ \lambda \end{bmatrix} = \begin{bmatrix} E & G^T \\ G & \mathbf{0} \end{bmatrix}^{-1} \begin{bmatrix} -l \\ h \end{bmatrix} \tag{16}$$

The KKT matrix is nonsingular. We use the following matrix M to represent the inverse of the KKT matrix:

$$\begin{bmatrix} M_{11} & M_{12} \\ M_{12}^T & M_{22} \end{bmatrix} = \begin{bmatrix} E & G^T \\ G & \mathbf{0} \end{bmatrix}^{-1}$$

Where $M_{11} \in \mathbb{R}^{S_2 \times (S_2+N)}$. We do not care about matrix M_{12} and M_{22} because M_{12} will multiply with a $\mathbf{0}$ vector (h vector is constituted by x_k and zero elements) and M_{22} produces the result of the dual variable λ, which is only required when constructing the KKT condition. In this way, only M_{11} is useful in our computation. This formulation of constructing the MPC formula reduces to the orignal storage requirement by $\frac{S1+S2}{S1-N}$, which is by 42.82% in the mass-spring system example in section V.

The solution to (9) is the result of a saturation function. The detailed processing steps are shown in Algorithm 1, which is pipelined in hardware. A common stopping criteria is to check if $\|\zeta^{i+1} - \zeta^i\|$ or $\|\upsilon^{i+1} - \upsilon^i\|$ is smaller than a certain value. We instead use a fixed number of iterations as the stopping criteria since for many control applications a relatively small number of iterations provides sufficient controller accuracy [7], and this reduces hardware complexity.

Algorithm 1: ADMM algorithm

1 Start from $i = 0$ with arbitrary ζ^0 and υ^0.
2 **do**
3 $\quad l := \begin{bmatrix} Q * R_k \\ \mathbf{0} \end{bmatrix} - \rho(\zeta^i + \upsilon^i)$ `// Update Vector l`
4 $\quad \chi^{i+1} := M_{11} * \begin{bmatrix} -l & x_k \end{bmatrix}^T$ `// Solve KKT`
5 $\quad \zeta^{i+1} := sat(\chi^{i+1} - \upsilon^i, \mathbf{dom}\, \mathcal{C})$ `// Saturation`
6 $\quad \upsilon^{i+1} := \upsilon^i + \rho(\zeta^{i+1} - \chi^{i+1})$ `// Update Dual`
7 $\quad i := i + 1$
8 **until** *stopping criterion is satisfied*;

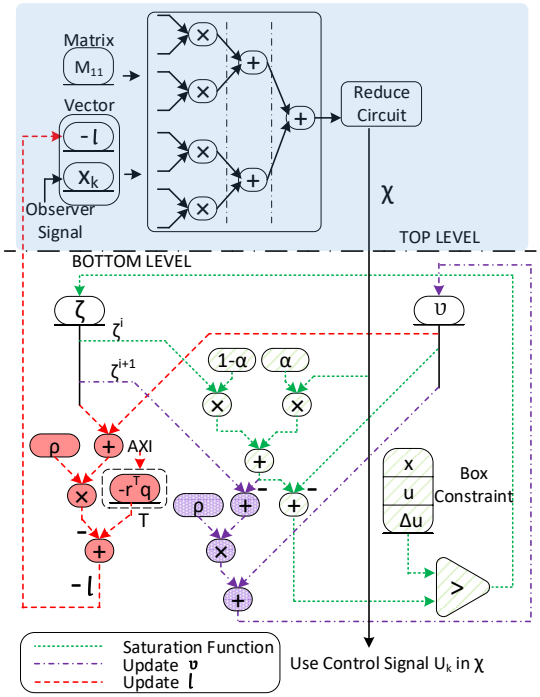

Fig. 1: Hardware Architecture for ADMM with Relaxation Parameter α.

As a final note on Algorithm 1, the χ^{i+1} appearing in line 5 and 6 is substituted with $\alpha\chi^{i+1} + (1-\alpha)\zeta^i$. $\alpha \in (0,2)$ is the relaxation parameter derived from Douglas-Rachford splitting. The convergence rate can be improved if α is properly selected.

IV. ADMM Hardware Architecture

This section introduces the detailed hardware architecture to support Algorithm 1. Major focuses of the design were system parametrization, system scaling, and runtime reference trajectory setting. Finally, we analyze the computation latency and BRAM usage. The design is written in VHDL using Vivado 2015.4 IDE.

A. ADMM Architecture Overview

As shown in Fig. 1, the high-level hardware architecture is divided into Top and Bottom level for a modularized illustration. Top level is the Quadratic Programming Solver. It has a matrix-vector multiplier tree structure. Bottom level is composed of three parts: 1) Saturation Function (marked in green diagonal line), 2) Update Vector υ^i (marked in purple cross line), 3) Update Vector l (marked in red). Some FIFOs which are used to store intermediate values are not shown in the figure. We use Block RAM(BRAM) to store the ζ^i, υ^i vector and the boundary value box constraints x, u, and Δu.

We next describe the components of Fig. 1 in greater detail:

1) Matrix-vector Multiplier (MVM): The QP solver is similar to the architecture presented in [20] for parallelizing MVM of large sparse matrices. The MVM computation uses a tree structure. D_p indicates the depth of the MVM tree. The total number of multipliers in the MVM tree is 2^{D_p} and the number of adders is $2^{D_p} - 1$.

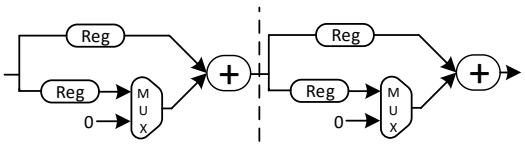

Fig. 2: Reduce Circuit Architecture with Two Cascaded Adders

2) Data Storage: Block RAMs (BRAMs) are used as the primary on-chip memory of the MPC engine. We configure the BRAMs as true dual port, which provides two read and two write ports. Every multiplier in the MVM tree has a dedicated dual port BRAM attached, with one port feeding matrix data and one port feeding vector data.

According to the BRAM configuration datasheet, the BRAM has to be at least 36Kb. Since the matrix size is the square of the vector size, we reserve a small fraction of space for vector storage. Another solution is to use Look Up Tables (LUTs) to store the vector. In either case, the vector should be duplicated to avoid write back conflicts.

We use N_{BRAM} to represent the number of available 36Kb BRAMs. N_{DSP} is the number of DSP slices on-chip. If the hardware resources meet inequality condition (17), the BRAM will hinder the scalability of the system without the support of the reduce circuit. This conclusion is based on the assumptions that 1) each multiplier and adder consume one DSP slice; 2) each MVM multiplier requires at least one 36kb BRAM. According to the Zynq datasheet, only the Z-7100 device from the Zynq-7000 family is unconformable to the inequality condition (17), which indicates that the on-chip memory is the resource bottleneck for most of the Zynq-7000 family.

$$2 * N_{BRAM} \geq N_{DSP} \geq 63.25\sqrt{N_{BRAM}} \tag{17}$$

3) Reduce Circuit: The purpose of the reduce circuit is to let us balance between resource usage and the number of MVM pipeline stages. Fig. 2 shows a reduce circuit structure, which is cascading two smaller reduce circuits.

The reduce circuit allows a single row of the matrix and vector to be separated into segments, and each segment is fed into the MVM pipeline in consecutive clock cycles. The reduce circuit accumulates the sum of each segment and generates a final result out of the last reduce stage. By employing more cascading levels, we can divide each row of the matrix into smaller segments at the cost of increasing latency due to increased pipeline stages at the reduce circuit. Suppose the depth of the MVM tree is D_p, and the number of M_{11} rows is N_{ROW} and columns is N_{COL}. Then the number of adders in the reduce circuit that our system requires is $N_R = \lceil N_{COL}/2^{D_p} \rceil - 1$. The number of clock cycles to merge all the matrix and vector data into the MVM pipeline is:

$$L_{read\_M_{11}} = N_{ROW} * (N_R + 1) \tag{18}$$

4) Saturation Function: Fig. 1 contains the hardware for saturation function. We assume each variable has same absolute upper and lower boundary value so that we just store the positive boundary values in the box constraint BRAM. The result of $\chi^{i+1} - \upsilon^i$ is compared with the box constraints. If the absolute value of $\chi^{i+1} - \upsilon^i$ is smaller than the box constraint, we output the value directly. Else, if the value is larger, we output the box constraint using the sign of $\chi^{i+1} - \upsilon^i$.

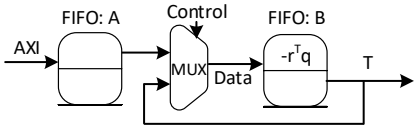

Fig. 3: Runtime Trajectory Planning

B. Trajectory Setting During Runtime

MPC can optimize a system's following of a trajectory multiple time steps ahead so that system can react accordingly in advance. Our controller can set the desired trajectory during runtime. For example, we may want to update a UAV's flight path while it is in the air.

We use two FIFOs to realize the functionality. The architecture is shown in Fig. 3, which is located in the dashed square marked by 'T' in Fig. 1. First, we configure FIFO:A and FIFO:B. FIFO:B stores the trajectory data for the current computation, and FIFO:A stores the future trajectory data. When computing the l vector, we read FIFO:B, use the data and write its output back to itself, which will be used for next converge iteration. In the last iteration, we discard the front N numbers after reading, and write the remaining back to FIFO:B. Next, we load N new trajectory data from FIFO:A into FIFO:B. In this way, each trajectory data shifts forward one sample step, and we fill the last trajectory point with a new one. Since writing to FIFO:A is independent of operations on FIFO:B, we can write new state trajectory data to FIFO:A at any time before FIFO:B goes empty.

C. Latency Analysis

Table I gives computation latency. The floating point multiplier latency is L_M=8; the floating point adder latency is L_A=11, the comparator latency is L_C=2. We call $L_{bt} + L_{bl}$ pure processing stages, namely the number of pipeline stages from an element entering the MVM pipeline to finishing.

TABLE I: Computation Latency

Binary Tree (L_{bt})	$L_M + D_p L_A + N_R(L_A + 2)$
Bottom Level (L_{bl})	$6L_A + 3L_M + L_C$

The total latency L_{ADMM} is shown in Equation. (19), which is the sum of pure processing stages and the clock cycles to finish fetching all the matrix and vector data into MVM tree ($L_{read\_M_{11}}$).

$$L_{ADMM} = L_{bt} + L_{bl} + L_{read\_M_{11}} \tag{19}$$

The architecture by default fetches one row per clock cycle or we can break each row into several pieces and accumulate each piece through the reduce circuit. However, under most cases, $N_{COL}/2^{D_p}$ is not an integer, thus some BRAMs store '0's to pad the final piece of the matrix row. These padding '0's occupy BRAM space and decrease the scalability of the

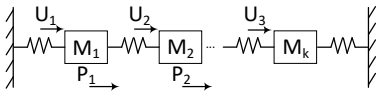

Fig. 4: Mass-spring System

design. The problem can be solved via introducing a simple state machine that tells the hardware which DSP should be fed '0' instead of reading data from BRAM.

V. Evaluation

This section describes our SW/HW co-design evaluation methodology. We evaluate our design using a Plant-on-Chip (PoC), which emulates the physical behavior of a linear system [21]. We provide post place&route results including resource usage and maximum clock frequency. We then compare our work with other related works.

A. Mass-spring System

The system we use to test our controller is a mass-spring model, which is considered as a benchmark in [15], [22] and many recent works use it to validate their hardware controller. The physical system is illustrated in Fig. 4. The objective is moving masses to desired positions by applying a force to each mass. The state vector consists of the position (P) and speed ($\dot{P}$) of each mass[2]. The state space model size increases quadratically as the number of masses increases. We constrain the position of each mass within 0.5m to avoid collision between adjacent masses. Each mass is 1Kg, and the spring constant is 1N/m. The input force (U) is limited to $\pm 0.5N$ and the change of input force between each sample period (i.e. rate of change, ΔU) is 0.1N, as in [18]. The sample period is 0.1s. The prediction horizon (H_p) and input horizon (H_u) are both 12. The cost constant for position P, input force U and input-rate ΔU are 80, 1 and 0 (we are not trying to minimize ΔU during the control process) respectively.

B. Plant on Chip Emulation

We conducted our experiments on a Zynq-7020 device. A Plant-on-Chip (PoC) was deployed onto the FPGA's Programable Logic (PL) fabric to emulate the mass-spring system shown in Fig. 4. The PoC executes state space Equations 1 and 2 with input u_k received from a hardware or software-based controller. The state of the PoC and control commands are logged out of band, using a UART interface, which is convenient for plotting the controller behavior at runtime.

The MPC controller requires the CPU to configure the co-processor BRAM content with the matrix M_{11}. When running the MPC hardware controller at 100 MHz, it computes u_k in 347.8μs using a D_p=3 MVM tree and executes 20 fixed converge iterations.

The hardware control graph is shown in Fig. 5. The red dashed line is the software configured trajectory, and the blue line is the actual mass position. The first 100 points of the trajectory are stored in the trajectory FIFO during

[2] P is the position relative to the initial position

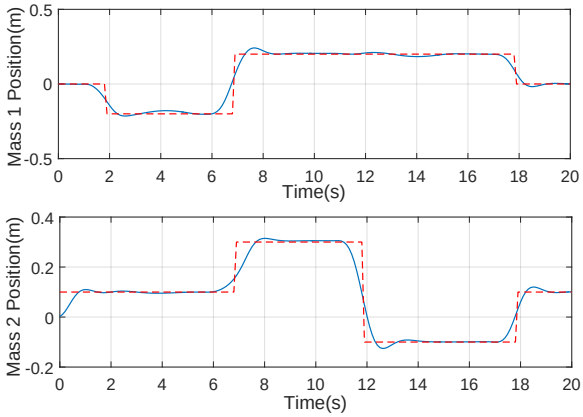

Fig. 5: Mass Position Change with respect to Planned Trajectory. Red dashed line is the planned trajectory, and the blue line is the actual trajectory.

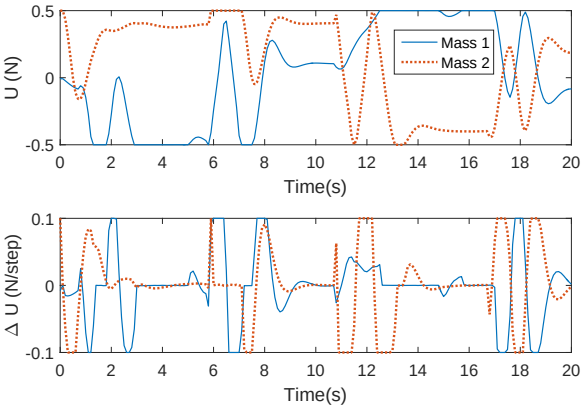

Fig. 6: Control Signal U and ΔU. Blue line is the input force and the force rate of change for M_1, red dashed line is for M_2.

configuration, which reflects 0s$-$10s of red trajectory line. The remainder of the trajectory is configured during runtime. The input force and the input force rate of change are shown in Fig. 6. From the graph, we can see that the control signal and control signal rate of change respect their bounding constraints of $\pm$0.5N and $\pm$0.1N per time step respectively.

C. SW/HW Co-design

The on-chip ARM processor transfers a pre-generated M_{11} matrix into the reconfigurable fabric's BRAMs through the AXI bus. Additionally, the on-chip ARM processor can access memory-mapped registers resident in the reconfigurable fabric to indicate when the MPC hardware engine should start and stop, and for updating the desired trajectory at runtime. The system block design is shown in Fig. 7. The design steps are:

1) According to system requirements, generate the bit-stream in Vivado, and M_{11} matrix in Matlab.
2) Store M_{11} to BRAM via AXI bus using ARM software.
3) ARM software configures trajectory and box constraints.
4) Send start signal to the PoC, and collect data through UART to external computer and plot graph.

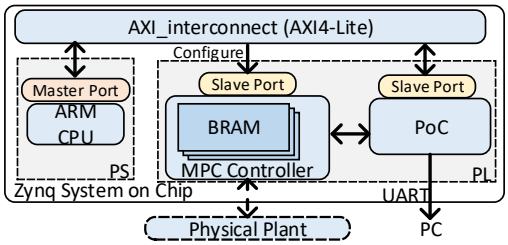

Fig. 7: Top Level System Overview

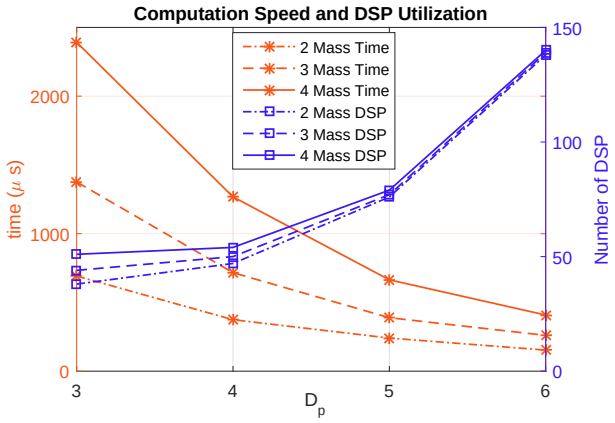

Fig. 8: Computation time of 40 converge iteration loops and DSP usage for different system configurations from simulation. Computation time is marked by $*$, number of DSPs is marked by $\square$. Hardware speed is 100MHz.

As we can see from the design steps, a control engineer can easily handle each step. In addition, when we apply the hardware control to another system with different parameters and size, the design can be easily adjusted.

D. Computation Speed Versus Hardware Resources

The relationship between DSP usage, D_p, and system size is shown in Fig. 8. As D_p increases, the size of the reduce circuit will decrease thus reducing computation time. The DSP resources used is modifiable. The more DSPs we use, the faster computation speed we achieve. Consider a situation where an FPGA has few DSP resources available, our proposed architecture can still execute the controller by decreasing the MVM depth, at the cost of increased computing time.

E. Resource Utilization and Timing Summary

Table II shows the resource utilization of the ADMM architecture. Each floating-point multiplier and adder use one DSP48E. We used the maximum number of pipeline stages supported by the Xilinx floating point adder(11) and multiplier(8) IP cores. Each memory attached to the MVM multiplier tree is composed of 2 36Kb BRAMs. BRAMs are also used for FIFOs in the MVM tree and Bottom level. The Zynq-7020 can hold an MVM tree having a depth up to 6. Generally, the clock frequency can easily reach 100MHz.

We also tested the resource scaling on a Zynq Ultrascale. It can reach 340MHz when deploying a $D_p = 8$ MVM tree.

TABLE II: Zynq-7020 Hardware Resource Usage

MVM Size D_p	Flip-Flops (106400 total)	LUTs (53200 total)	18Kb BRAM (280 total)	DSP48E (220 total)	Maximum Frequency
3	18147	12746	55	38	151.149MHz
4	21058	15103	87	47	144.885MHz
5	32425	23391	151	76	143.699MHz
6	57167	41273	279	138	133.298MHz

F. Comparision with other Works

Table III provides a comparative summary of our work with two other FPGA accelerated MPC works and a software implementation. For the hardware work, one is IPM based [15], and another is ADMM based [18], like ours. Starting with the IPM based work, it can be seen when compared against the ADMM-based approaches using about half the number of multipliers (˜200) for about the same number of decision variables (˜200) our approach is about 50x faster (2,650us vs. 46.1us) than the IPM implementation and the other ADMM work [18] is about 100x faster (2,650us vs. 23.4us). As explained in Section II, this is due to the ADMM algorithm requiring less computation per convergence iteration. The software implementation, [7], is ADMM based, and is run on a 3.4GHz Xeon processor. We estimate for 262 variables that this SW solution would take 850 μs if the average number of iteration maintains in 35.1.

When comparing our approach with the ADMM based approach of [18], for about 200 multipliers and 200 decision variables, it can be seen that our approach is about two times slower (i.e. 46.1 us vs. 23.4 us). The primary reason for this is that in [18] fixed point arithmetic is used, while our implementation uses 32-bit floating point arithmetic. The main way in which this impacts performance is that the floating point adders required 11 pipelining stages to maximize clock frequency, while fixed point addition can be done at a high clock rate in one clock cycle. For the size of matrices being operated on (limited by on-chip memory), the number of adder pipeline cycles to fill the processing pipeline (˜88 for an MVM tree of depth 8) is nearly half the number of matrix rows (˜200) read into the MVM tree. This accounts for a vast majority of the two times difference in performance. However, for this cost in performance, we gain the convince of software and control algorithm developers not having to deal with the complexities of working with custom fixed-point number formats, easing the process of deploying a designed controller into our MPC accelerator. For physical systems requiring sub-millisecond controller updates rates, this is a good tradeoff, however for controllers requiring 10s of microsecond updates rates using a fixed-point approach is more appropriate with todays FPGA capabilities. The important question to answer is for your application is the convenience of using floating point worth the performance tradeoff.

The Reduce circuitry implemented in our architecture naturally allows our design to scale to large numbers of decision variables using a nearly arbitrarily small number of multipliers at the cost of speed, as is illustrated in the 350* entry of

TABLE III: Hardware Computation Time per Iteration between Related Work.[3]

	Method	Data Format	Chip Series	f_{clk}	#Multipliers	Iteration	#Opt Var	Running Time
This Paper	ADMM	floating-point	Zynq-7020	130MHz	72 (D_p=6, K=1)	40	204	314.2 μs
							350*	717.2 μs
					80 (D_p=5, K=2)		204	291.4 μs
			ZU9EG (Zynq UltraScale+)	340MHz	264 (D_p=8, K=1)			46.1 μs
					792 (D_p=8, K=3)			30.1 μs
HW [18]	ADMM	fixed-point	Virtex-6 (LX75)	400MHz	216 (K=1)	40	216	23.4μs
			Virtex-6 (SX475)		1512 (K=7)			4.90μs
HW [15]	IPM	floating-point	Virtex-7 (XC7VX485T)	200MHz	448	10	240	2,650 μs
SW [7]	ADMM	floating-point	Quad-core Intel Xeon	3.4GHz	n/a	35.1	525	3,400 μs

Table III, while [18] does not have such a mechanism for scaling the number of decision variables above the number of multipliers in the system. This gives our MPC engine the flexibility to be deployed into System-on-Chip FPGA applications that may not have many multipliers to allocate to an MPC computation engine. Two ADMM tuning features implemented in our ADMM architecture that is not implemented by [18] is a relaxation parameter (α) and a dual update step length (ρ), which can be used to tune the convergence rate of ADMM for a given system.

A final point of comparison is that this work tightly integrates an on-chip ARM processor with the MPC compute engine. This provides Controls or Software engineers a convenient software mechanism for configuring the controller for arbitrary systems, updating the desired trajectory of the system at runtime, and tuning the ADMM α and ρ parameters.

VI. Conclusion

We have presented an MPC acceleration engine that is tightly coupled to an ARM processor embedded on the same chip. Our acceleration engine has been designed to allow trading off between performance and hardware resource usage. Our tight interface with an on-chip ARM processor allows software to easily update the configuration of the acceleration engine to control a wide range of systems, and to adjust the desired system trajectory at run-time. An avenue of future work is examining the architectural details required for interfacing and managing external sensors to extend our evaluation from controlling a real-time Plant on Chip to actual physical systems, such as quadcopters.

References

[1] A. S. Kumar and Z. Ahmad, "Model predictive control (MPC) and its current issues in chemical engineering," *Chemical Engineering Communications*, vol. 199, no. 4, 2012.

[2] S. Kouro, P. Cortes, R. Vargas, U. Ammann, and J. Rodriguez, "Model predictive control–a simple and powerful method to control power converters," *IEEE Transactions on Industrial Electronics*, vol. 56, no. 6, June 2009.

[3] T. Tettamanti, I. Varga, B. Kulcsar, and J. Bokor, "Model predictive control in urban traffic network management," in *2008 16th Mediterranean Conference on Control and Automation*, June 2008.

[4] A. Richards and J. How, "Decentralized model predictive control of cooperating UAVs," in *43rd IEEE Conference on Decision and Control*, vol. 4, Dec 2004.

[3]K indicates the number of times the core MPC engine with D_p MVM tree is duplicated to process Matrix rows in parallel

[5] "Model predictive heuristic control: Applications to industrial processes," *Automatica*, vol. 14, no. 5, 1978.

[6] J. M. Maciejowski, *Predictive control with constraints*. Essex, England: Prentice Hall, 2002.

[7] B. O'Donoghue, G. Stathopoulos, and S. Boyd, "A splitting method for optimal control," *IEEE Transactions on Control Systems Technology*, vol. 21, no. 6, Nov 2013.

[8] H. Kwakernaak, *Linear Optimal Control Systems*, R. Sivan, Ed. New York, NY, USA: John Wiley & Sons, Inc., 1972.

[9] F. Borrelli, A. Bemporad, and M. Morari, "Predictive control for linear and hybrid systems, 2015," *preparation, available online at http://www. mpc. berkeley. edu/mpc-course-material*, 2015.

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

[11] J. Nocedal and S. J. Wright, *Numerical Optimization*, 2nd ed. New York: Springer, 2006.

[12] S. Boyd and L. Vandenberghe, *Convex Optimization*. New York, NY, USA: Cambridge University Press, 2004.

[13] J. L. Jerez, G. A. Constantinides, and E. C. Kerrigan, "FPGA implementation of an interior point solver for linear model predictive control," in *Field-Programmable Technology, International Conference on*, Dec 2010.

[14] G. Li, J. Gu, Q. Song, Y. Lu, and B. Zhou, "The hardware design and implementation of a signal reconstruction algorithm based on compressed sensing," in *Intelligent Networks and Intelligent Systems, Fifth International Conference on*, Nov 2012.

[15] J. Liu, H. Peyrl, A. Burg, and G. A. Constantinides, "FPGA implementation of an interior point method for high-speed model predictive control," in *24th International Conference on Field Programmable Logic and Applications*, Sept 2014.

[16] A. Wills, A. Mills, and B. Ninness, "FPGA implementation of an interior-point solution for linear model predictive control," *IFAC Proceedings Volumes*, vol. 44, no. 1, 2011.

[17] M. S. K. Lau, S. P. Yue, K. V. Ling, and J. M. Maciejowski, "A comparison of interior point and active set methods for FPGA implementation of model predictive control," in *Control Conference, European*, Aug 2009.

[18] 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, 2014.

[19] T. V. Dang, K. V. Ling, and J. M. Maciejowski, "Embedded ADMM-based QP solver for MPC with polytopic constraints," in *Control Conference, European*, July 2015.

[20] L. Zhuo and V. K. Prasanna, "Sparse matrix-vector multiplication on FPGAs," in *Proceedings of the 2005 ACM/SIGDA 13th International Symposium on Field-programmable Gate Arrays*.

[21] S. Vyas, C. Kumar, J. Zambreno, C. Gill, R. Cytron, and P. Jones, "An FPGA-based Plant-on-Chip platform for cyber-physical system analysis," *IEEE Embedded Systems Letters*, vol. 6, no. 1, 2014.

[22] 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, 2011*.