FPGA Implementation of an Interior Point Method for High-speed Model Predictive Control

Junyi Liu*, Helfried Peyrl†, Andreas Burg‡, George A. Constantinides*
*Department of Electrical and Electronic Engineering, Imperial College London, SW7 2AZ, United Kingdom
{junyi.liu13, g.constantinides}@imperial.ac.uk
†ABB Corporate Research, Segelhofstrasse 1K, 5405 Baden-Dattwil, Switzerland
helfried.peyrl@ch.abb.com
‡Telecommunication Circuits Laboratory, Ecole Polytechnique Federale de Lausanne (EPFL), Switzerland
andreas.burg@epfl.ch

***Abstract*—In this paper, we present a hardware architecture for implementing an interior point method for model predictive control (MPC) on field programmable gate arrays (FPGA). The FPGA implementation allows the solution of quadratic programs occurring in MPC at very high speed. Experiments show that our hardware implementation is able to outperform an software implementation running on a high-end CPU while consuming significantly less power making it well-suited for embedded industrial control applications. In contrast to existing FPGA implementations, the proposed solution exploits the MPC-specific problem structure with the direct linear equation solver and uses an efficient predictor-corrector algorithm. Moreover, the modular design of the architecture simplifies customization or extension to special control problem classes. The proposed FPGA solution can broaden the applicability of solving complex or large MPC problems in embedded computing platforms that were so far considered out of reach.**

I. INTRODUCTION

Model predictive control (MPC) represents one of the most important paradigms in the control of industrial applications. Although there exist many different variants of MPC, they all share the same basic ingredients: a *model* to predict the evolution of the the physical state of the controlled plant over time, an *optimization procedure* which computes the control action in accordance with the control objectives, and a *receding horizon* strategy. The control objectives are encoded in the objective function and the constraints of an optimization problem whose decision variables are the future control actions (inputs) and the corresponding predicted plant states. More specifically, given the current measured (or estimated) plant state, the optimization problem uses a discrete-time model to predict the evolution of the plant's state over a *finite prediction horizon*. The solution of the problem is a finite sequence of optimal control inputs. Although a control strategy for the next time steps is then readily available, only the first control move gets applied to the plant. In the next sampling time instant the entire procedure is repeated, *i.e.* the current state is measured to compensate for modeling errors and disturbances, and the optimization problem is solved for the shifted horizon. Consequently, the problem is solved *on-line* at each sampling instant, typically by using an iterative algorithm. For more detailed information on MPC, the reader is referred to [1], [2].

Until recently, MPC has been mainly restricted to processes with rather slow dynamics and with sampling times ranging from a few minutes to many hours, such as the ones encountered in the areas of processing (petro)chemicals, minerals and metals. The main reason for this restriction can be traced to the huge computational demand that optimization-based algorithms pose to the computational hardware. However, the ever increasing computational capacity as well as algorithmic advances enabled MPC implementations that achieve solution times in the sub-second range. Examples of these efforts are the MPC-tailored algorithms proposed in [3]–[5].

While MPC problems are traditionally solved with high performance general purpose CPUs, a rather recent trend is to employ FPGA-based implementations to push the limits of MPC even further, see, *e.g.* [6]–[11]. These solutions render MPC feasible for embedded control applications that require solution times in the milli- or even micro-second range and a deployment of the control algorithm on a low power hardware platform.

Our objective is an FPGA implementation of a high-speed solver for mid-scale MPC problems. In particular, our hardware implementation is based on the MPC-structure exploiting interior point method presented in [5]. In the scope of this work, the algorithm is analyzed with the goal to realize a fast and flexible FPGA implementation. We make the following contributions to advance the state-of-the-art in FPGA-based MPC solvers:

- To the best of our knowledge, the presented solution is the first hardware implementation of an interior point solver that exploits the inherent problem structure of MPC in the context of a predictor-corrector scheme. The linear system underlying the optimization procedure is solved directly using a block-wise Cholesky factorization to support the efficient implementation of the predictor-corrector scheme.
- The solution is a parameterizable hardware architecture that can be adapted to the problem dimensions. The modular design based on systolic arrays can be easily extended/customized to more generic or specialized MPC problem classes. Examples of these are exploitation of simple box constraints or support for quadratic inequality

constraints.

The rest of this paper is organized as follows. In Section II, we compare our solution with other approaches and previous FPGA implementations to motivate our work. The target optimization algorithm is described briefly with emphasis on some of its specific features in Section III. In Section IV the hardware architecture is described from a system-level design point of view and the core modules are presented. The analysis and experimental results are provided in Section V. Finally, conclusions are summarized in Section VI.

II. Related Work

A linear quadratic MPC problem is typically formulated as a quadratic programming (QP) problem which is then solved at each sampling instant using an iterative procedure. Algorithmically, one usually distinguishes between interior point methods (IPM), active-set methods (ASM), and first-order (gradient-based) methods.

Despite their simplicity, first-order methods only recently gained increased attention in the control community with the application of Nesterov's fast gradient method to input-constrained problems in [3]. Extensions to more generic MPC problems were then proposed by several authors and high-speed FPGA implementations in fixed-point arithmetic were presented in [10], [11]. Although the reported execution times of these implementations are indeed impressive, they are limited to simple constraints (such as box constraints modeling actuator saturation). Moreover, their sensitivity with respect to problem conditioning, and the sub-linear convergence rate in the dual domain, limits the applicability of gradient-based methods to generic and/or badly conditioned MPC problems that are not uncommon in practice.

Both ASM and IPM solvers have a long track record of successful application in the area of MPC. Compared to the worst case exponential complexity of ASMs, the computational effort of IPMs grows only polynomially [12]. ASMs are typically applied to a problem formulation in which the equality constraints have been eliminated to yield a smaller but dense QP. Generally, problems where the number of states is relatively large and the prediction horizon is long, favour a structure exploiting IPM [13]. This is also concluded in [6], where the authors compare the two approaches from an FPGA implementation point of view. Moreover, the latter paper reports an increased numerical robustness of IPMs in comparison with ASMs.

The computational bottleneck of IPMs is caused by solving a linear system of equations in every iteration of the algorithm. The equations can be solved either directly by factorization and substitution, or indirectly by iteratively improving an approximation of the solution. Iterative approaches are generally advantageous for large, sparse systems. Moreover, they require only basic mathematical operations and exhibit a relatively simple data flow which renders them well-suited for hardware implementations. Not surprisingly, recent IPM implementations for MPC are based on indirect solution methods. The architecture proposed in [9] applies a primal barrier method to a dense QP formulation that can handle actuator box constraints. The underlying linear system is solved using the conjugate gradient method. The applicability of the implementation, however, is limited by its restriction to box constraints and the dense problem formulation which renders the approach unfavourable for larger prediction horizons. The FPGA implementation in [7] uses the minimum residual (MINRES) method to solve the indefinite linear system arising in a primal-dual method. Although the approach takes advantage of sparsity by using compressed diagonal storage for the banded linear equation matrix, the inherent MPC problem structure is not explicitly exploited for a predictor-corrector scheme.

In contrast to iterative approaches, direct methods can efficiently solve the same system of equations for different right hand-side vectors once the matrix factorization is computed. This makes them well-suited for Mehrotra's predictor-corrector IPM which requires significantly fewer outer iterations than conventional IPMs [14].

MPC-structure exploiting IPMs with direct linear equation solvers have initially been presented in [4] which employs a block-wise Cholesky factorization to a primal IPM. The paper inspired the development of the primal-dual solver `FORCES` [5], a C code generator based on Mehrotra's algorithm that can handle generic multi-stage control problems and even supports quadratic inequalities. The generated code does not only take advantage of the block-banded MPC structure but also exploits problem specific properties such as diagonal cost matrices or box constraints. The result is a highly efficient code that has been proven to outperform many competing top-class solvers [5].

Based on the efficient `FORCES` algorithm, we propose an FPGA implementation, which is composed of multiple systolic-array based pipelines. It is able to achieve faster execution speed than the software version of `FORCES` and the FPGA-based IPM solver from [7]. Moreover, its computation resources usage is not related to the length of the horizon in the MPC problem. Further attractive features of our IPM implementation will be discussed in Section IV.

III. Optimization Algorithm

Our target MPC problem can be cast into a more general multi-stage optimization problem as shown below.

$$
\begin{aligned}
\min_{y_i, i=1\ldots N} \quad & \sum_{i=1}^{N} \frac{1}{2} {y_i}^T H_i y_i + \mathrm{f}_i^T y_i \\
\text{s.t.} \quad & g_i(y_i) \leq 0 \,, \quad i = 1, \ldots, N \,, \\
& L_1(y_1) = 0 \,, \\
& L_i(y_i, y_{i-1}) = 0 \,, \quad i = 2, \ldots, N \,,
\end{aligned} \tag{1}
$$

where $y_i := [x_i^T, u_i^T]^T \in \mathbb{R}^{n_y}$ denote the stage variables comprising predicted plant states $x_i \in \mathbb{R}^{n_x}$ and control inputs $u_i \in \mathbb{R}^{n_u}$. The objective function is given by positive semidefinite cost matrices $H_i \in \mathbb{R}^{n_y \times n_y}$ and affine vectors $\mathrm{f}_i \in \mathbb{R}^{n_y}$. In the context of MPC, the number of stages N is referred to as prediction horizon, and x_1 is equal to the measured (or estimated) current state.

We assume that the inequality constraint functions $g_i \colon \mathbb{R}^{n_y} \mapsto \mathbb{R}^{n_{ineq}}$ are linear and that the feasible set has non-empty interior. Throughout this paper, we assume that g_i are affine functions. The equality constraint functions $L_i \colon \mathbb{R}^{n_y} \mapsto \mathbb{R}^{n_x}$ are given by:

$$\begin{aligned} L_1(y_1) &:= D_1 y_1 + c_1 \ , \\ L_i(y_i, y_{i-1}) &:= C_{i-1} y_{i-1} + D_i y_i + c_i \ , \quad i = 2, \dots, N \ , \end{aligned}$$

where $C_{i-1} \in \mathbb{R}^{n_x \times n_y}$, $D_i \in \mathbb{R}^{n_x \times n_y}$, and $c_i \in \mathbb{R}^{n_x}$ describe the discrete-time dynamics of the controlled plant.

Mehrotra's primal-dual IPM [14] has been shown to be highly efficient to solve the Karush-Kuhn-Tucker (KKT) optimality conditions associated with (1) with Newton's method [15]. The algorithm details are presented in [12]. Here, we introduce Mehrotra's predictor-corrector primal-dual IPM following the description in [5] and [12].

Linearizing the KKT conditions of (1) yields the following structured linear equation system:

$$\begin{bmatrix} \mathcal{H} & C^T & J^T(y) & \\ C & & & \\ J(y) & & & I \\ & & S & \Lambda \end{bmatrix} \begin{bmatrix} \Delta y \\ \Delta v \\ \Delta \lambda \\ \Delta s \end{bmatrix} = - \begin{bmatrix} r_{\mathrm{C}} \\ r_{\mathrm{E}} \\ r_{\mathrm{I}} \\ r_{\mathrm{s}} \end{bmatrix} \ , \quad (2)$$

where

$$C := \begin{bmatrix} D_1 & 0 & 0 & \dots & 0 \\ C_1 & D_2 & 0 & \dots & 0 \\ 0 & C_2 & D_3 & \dots & 0 \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ 0 & 0 & \dots & C_{N-1} & D_N \end{bmatrix} , \quad J(y) := \begin{bmatrix} \nabla g_1(y_1) & 0 & \dots & 0 \\ 0 & \nabla g_2(y_2) & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & \nabla g_N(y_N) \end{bmatrix} ,$$

$$S := Is, \ \Lambda := I\lambda,$$

and $\mathcal{H} := \left[H_1^T, \dots, H_N^T\right]^T$, $v := \left[v_1^T, \dots, v_N^T\right]^T$, $s := \left[s_1^T, \dots, s_N^T\right]^T$, and $\lambda := \left[\lambda_1^T, \dots, \lambda_N^T\right]^T$. $s_i \in \mathbb{R}_{\geq 0}^{n_{ineq}}$ are slack variables introduced for the inequality constraints. $v_i \in \mathbb{R}^{n_x}$ and $\lambda_i \in \mathbb{R}_{\geq 0}^{n_{ineq}}$ are dual variables associated with the equality and inequality constraints, respectively. I denotes the identity matrix. If some variables have smaller size in some stages, they are padded with zeros to allow fixed-size variables in all stages.

A. *Predictor-corrector IPM*

Due to limited space, only the main steps and critical matrix operations of the Mehrotra's IPM (see Algorithm 1) are introduced in the following.

As illustrated in Algorithm 1, in each iteration the linear system (2) is solved twice with the same coefficient matrix $\mathcal{A}$ but different right-hand sides: first for the so-called *affine-scaling step* (line 4), and second for the *centering-corrector step* (line 7), as described in [12]. Once the search direction Δz_{aff} is computed, a line search is performed to determine the maximum updating step size along this direction. The second right-hand vector b_{cc} is generated based on the direction Δz_{aff} and step size α_{aff} obtained in the affine-scaling step. After the centering-corrector step is finished, the current iterate z is updated using the combined direction Δz and the final step size α which is again obtained from a line search.

Algorithm 1 Predictor-corrector primal-dual IPM

Require: $z \leftarrow (y, v, \lambda, s)$, choose initial point z_0 with $v_0 \leftarrow 1, \lambda_0 \leftarrow 1$

1: **repeat**
2: Generate coefficient matrix $\mathcal{A}$
3: Generate right-hand vector b_{aff} for affine-scaling step
4: Solve $\mathcal{A} \Delta z_{\text{aff}} = b_{\text{aff}}$
5: $\alpha_{\text{aff}} \leftarrow \max(0, 1], \text{ s.t. } \begin{bmatrix} \lambda + \alpha_{\text{aff}} \Delta \lambda_{\text{aff}} \\ s + \alpha_{\text{aff}} \Delta s_{\text{aff}} \end{bmatrix} > 0$
6: Generate right-hand vector b_{cc} for center-corrector step
7: Solve $\mathcal{A} \Delta z_{\text{cc}} = b_{\text{cc}}$
8: $\Delta z \leftarrow \Delta z_{\text{aff}} + \Delta z_{\text{cc}}$
9: $\alpha \leftarrow \max(0, 1], \text{ s.t. } \begin{bmatrix} \lambda + \alpha \Delta \lambda \\ s + \alpha \Delta s \end{bmatrix} > 0$
10: $z \leftarrow z + \alpha \Delta z$
11: **until** (convergence criteria fulfilled or maximum iteration number reached)

The computation of the search direction comprises generating a potentially large matrix and solving the resulting linear equation systems. The corresponding computations are the performance bottleneck in any interior point method and a fast implementation of these operations is crucial for the application of MPC to high-speed applications, see, *e.g.*, [4].

B. *Block-wise linear solver*

As in the `FORCES` code, the inherent block-banded problem structure of MPC is exploited in our implementation when solving the linear systems in lines 4 and 7 in Algorithm 1: all linear algebra kernel operations are performed on small blocks instead of the full matrix. Elimination of the variables Δy, $\Delta \lambda$ and Δs from (2) yields the so-called *normal equations* [15] while still preserving the block-banded structure.

$$Y \Delta v = \beta \ . \quad (3)$$

The new coefficient matrix Y in (3) is symmetric positive definite with tri-diagonal block-banded structure and thus can be factorized efficiently using a block-wise Cholesky decomposition as described in [5]: $Y = L_Y L_Y^T$, where

$$Y := \begin{bmatrix} Y_{d1} & Y_{sd1} & 0 & \dots & 0 \\ Y_{sd}{}_1^T & Y_{d2} & Y_{sd2} & \dots & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & 0 & Y_{sd}{}_{N-2}^T & Y_{dN-1} & Y_{sdN-1} \\ 0 & 0 & 0 & Y_{d}{}_{N-1}^T & Y_{dN} \end{bmatrix} , \quad L_Y := \begin{bmatrix} L_{d1} & 0 & 0 & \dots & 0 \\ L_{sd1} & L_{d2} & 0 & \dots & 0 \\ 0 & L_{sd2} & L_{d3} & \dots & 0 \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ 0 & 0 & \dots & L_{sdN-1} & L_{dN} \end{bmatrix} ,$$

and

$$Y_{d1} = L_{d1} L_{d}{}_1^T \ , \quad (4a)$$

$$Y_{di} - L_{sdi-1} L_{sd}{}_{i-1}^T = L_{di} L_{d}{}_i^T \ , \quad 2 \leq i \leq N \ , \quad (4b)$$

$$Y_{sd}{}_i^T = L_{sdi} L_{d}{}_i^T \ , \quad 1 \leq i < N - 1 \ . \quad (4c)$$

$L_{di} \in \mathbb{R}^{n_x \times n_x}$ are lower triangular and Y_{di}, Y_{sdi}, $L_{sdi} \in \mathbb{R}^{n_x \times n_x}$ are dense matrices. Equations (4a) and (4b) are solved with Cholesky decompositions with the respective matrices on the left hand side. In Equation (4c), L_{sdi} is obtained by matrix substitution, which is realized by the vector forward substitutions for the row vectors of $Y_{sd}{}_i^T$.

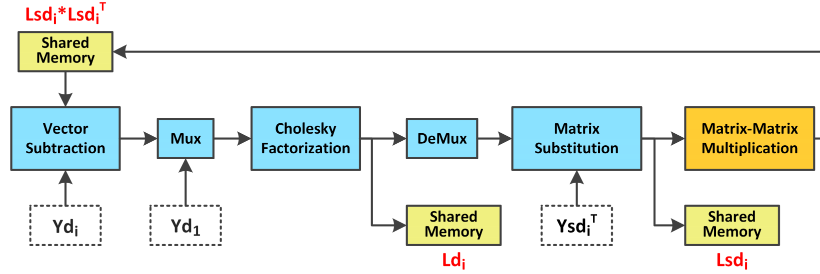

Fig. 1. Block diagram of a systolic-array based pipeline for the block-wise Cholesky factorization in (4).

Using (4a)-(4c), Y can be factorized in an iterative process that only involves N Cholesky decompositions of relatively small matrices of size n_x. Each block corresponds to the sub-problem for one horizon step of the entire problem in (3). Since L_{di} depends on L_{sdi-1}, there is a data dependency between two neighboring sub-problems. Therefore, the entire factorization is coupled via (4b) and thus these sub-problems must be solved successively.

To solve the entire problem efficiently, we can apply various levels of pipelining in the hardware design to accelerate the algorithm. Solving (3) can be separated into three main computation steps: generation of Y, factorization of Y, and solution of $L_Y L_Y^T \Delta v = \beta$ by vector forward and backward substitutions. Each of these steps processes the data of one sub-problem after the other, and together they form a *stage-level pipeline*. Moreover, there exist parallelization and pipelining opportunities at a more fine-grained level as well. For example, the factorization stage can be realized by a *column-level pipeline*: the calculation of L_{di} has an intrinsic column-wise data dependency, so that L_{sdi} can be calculated column-wise to achieve same data processing rate. The corresponding computation structure will be discussed in more detail in Section IV.

IV. Hardware Implementation

A. Number representation

An important design choice for the implementation of an algorithm is the employed number representation, *i.e.* fixed-point or floating-point. The algorithm at hand contains various operations including division, matrix substitution and Cholesky factorization. Since in IPMs the linear equation system to be solved in each iteration becomes ill-conditioned as the solution of the optimization problem is approached, a number representation with a high dynamic range is preferable. Therefore, we currently use single precision floating point numbers as most of the existing FPGA implementations of IPMs.

Our current target FPGA platform is the Xilinx Virtex7, and its floating-point unit (FPU) IPs are used for the floating-point computations. In the Virtex7, Xilinx IPs adopt ARM's AXI interface [16], which is also used in the system design. The FPU IPs support all necessary floating-point operations and the provided parametrization allows to trade-off resource usage and performance in a wide range.

B. Systolic-array based pipelines

The full algorithm comprises dozens of matrix and vector operations that also have complex data dependencies. To mitigate the complexity of the full architecture, our design is based on the systolic array architecture template [17]. Systolic arrays were firstly applied to high performance multi-processor systems, and are now widely used in ASIC design for signal processing, for example in digital wireless communication.

The hardware system is partitioned into five systolic-array based pipelines. Their main functions are summarized as follows.

- Generate matrix Y and its factor matrix L_Y in (3).
- Generate vector β in (3).
- Solve (3).
- Generate $(\Delta v, \Delta\lambda, \Delta s)$.
- Compute the search step size α and update vector z.

These pipelines also support parallel and pipelined execution at the stage level as long as is permissible by the algorithm. For example, the calculations for matrix Y and vector β are independent of each other, so that the first two pipelines work fully in parallel. The data flow among all pipelines is realized via shared memory modules to buffer and reuse data. In each pipeline, there are many concatenated computation, memory, and switch modules. For each module, we apply the simplified handshake interface compatible to the AXI4-stream protocol [18] to enable spontaneous data transfer. Inside each module, one finite state machine controls the interface and data processing circuits.

The modular design simplifies the integration of problem-tailored or extended computation modules. For example, we designed specialized modules that benefit from the computational simplifications resulting from box constraints. Similarly, an extension of the same modules to support quadratic inequalities instead of linear ones as described in [5] is also conceivable.

Fig. 1 shows the critical part of the pipeline which implements the block-wise Cholesky factorization algorithm described in (4). This partial pipeline will serve as a running example to further illustrate our hardware implementation. The

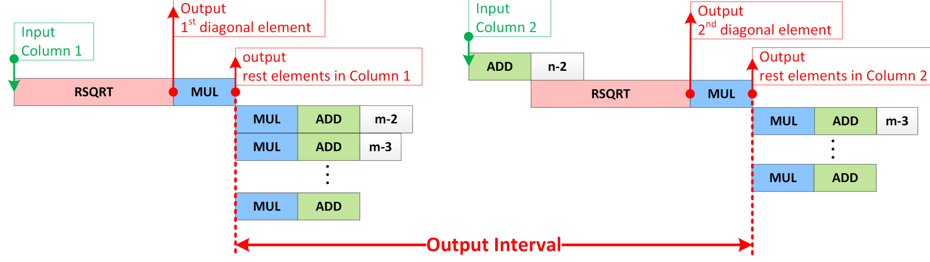

Fig. 2. Timing diagram of the hardware implementation of Cholesky factorization.

arrows connecting the modules depict the data transfer using the handshake interface.

The matrix computations in (4a) and (4b) are mapped into the modules called *Vector Subtraction*, *Cholesky Factorization* and *Matrix-Matrix Multiplication*. Likewise, the *Matrix Substitution* module implements (4c). The input data are fed into these modules with different throughput rates ranging from a single element of a vector over a column of a matrix to an entire matrix per cycle. After a known delay, the output data will be collected by memory modules and/or fed into another modules. The delay of each pipeline module (cycle cost) is affected by the inner parallelism of the modules, which also affects the throughput of their interfaces. Especially for matrix computations, higher parallelism implies more resource usage but faster computation in the hardware implementation.

C. Computation modules

In the pipelined architecture, there is no need to always aim at the highest parallelism in all modules, since the pipeline throughput is determined by the slowest or the most time consuming module in the pipeline. Computation modules consume most of the time to perform the linear algebra operations. Since minimum delay is the main criterion for an effective MPC solver and the Cholesky decomposition constitutes the computationally most demanding operation in the IPM, our design aims at a fast Cholesky implementation by fully exploiting its parallelization potential. The level of parallelism and the throughput of this module essentially dictates the inner parallelism of the other modules to match the throughput with minimum resource usage without affecting the entire system performance.

Because of its important role in the algorithm, the implementation of the Cholesky factorization will be discussed in the following in more detail.

1) Background: The Cholesky factorization is used to decompose a symmetric positive definite $n \times n$ matrix A in such a way that $A = LL^T$, where $L \in \mathbb{R}^{n\times n}$ is a lower triangular matrix that is computed using the following equations:

$$\begin{aligned} L_{j,j} &:= (A_{j,j} - \sum_{k=1}^{j-1} L_{j,k}^2)^{\frac{1}{2}}, \\ L_{i,j} &:= \tfrac{1}{L_{j,j}}(A_{i,j} - \sum_{k=1}^{j-1} L_{i,k}L_{k,j}), \quad i > j. \end{aligned}$$

It follows from these equations that L can be calculated column-wise from left to right, since the calculation of one column depends on all the columns calculated before it. Therefore, the Cholesky decomposition has intrinsic column-wise data dependencies. In the following we describe the details of our implementation.

We replace the square root (SQRT) and divide (DIV) operations in the computation of $L_{j,j}$ by reciprocal square root (RSQRT) and multiply (MUL). This change of operations is justified by the fact that the factor matrices are only used for forward/backward substitution where the necessary division can then be substituted by a multiplication. Since the Xilinx FPU [19] allows realization of the RSQRT and MUL operations with a smaller computational delay and at the same time fewer logic resources than the SQRT and DIV operations, the modified Cholesky factorization is better suited for high-speed and efficient implementation:

$$\begin{aligned} \tilde{L}_{j,j} &:= (A_{j,j} - \sum_{k=1}^{j-1} L_{j,k}^2)^{-\frac{1}{2}}, \\ L_{i,j} &:= \tilde{L}_{j,j}(A_{i,j} - \sum_{k=1}^{j-1} L_{i,k}L_{k,j}), \quad i > j. \end{aligned} \tag{5}$$

2) Architecture: The hardware implementation of the Cholesky factorization uses a column-wise computing structure following the corresponding data dependency mentioned before. There are $n-1$ parallel *element-level pipelines* corresponding to the computation of intermediate results for 2nd to n^{th} columns of matrix L, because the calculation of the first column of matrix L just needs the first input column of matrix A. Each *element-level pipeline* has one multiplier, one adder and an additional register chain. These components work in pipeline to calculate partial values based on the the previous column results, corresponding to the operations in (5). In addition, there is one FPU for RSQRT for the calculation of the diagonal elements of L. To match the pipeline throughput rate applied in FPUs, the *element-level pipelines* for each result column have the same throughput rate of 1 number/cycle.

Figure 2 shows the timing diagram of calculating the first two columns of L. The latency of RSQRT, MUL, ADD are defined as $t_{rsqrt}, t_{mul}, t_{add}$ respectively. We assume $n > t_{add}$ and define $m = n - t_{add}$. The minimum output interval between two columns is given by $t_{rsqrt}+2t_{mul}+2t_{add}$, which can be used to evaluate the latency of the parent systolic-array pipeline building block.

D. Data flow control

Since the data flow of our target algorithm is very complex, it is not convenient and scalable to control this data flow by a central finite state machine. Because the spontaneous data transfer interfaces are applied, the control of data flow is distributed into the configuration of these interfaces, switch modules and memory modules.

1) Multi-target handshake: To realize the data flow from one computation or memory module's output port to multiple input ports, the output port is enhanced with the support of multi-target handshaking signal pairs. For instance, in Fig. 1, the output data of *Matrix Substitution* is fed into *Matrix-Matrix Multiplication* and one memory module. The number of handshaking ports of one output interface is programmable to support more than two input interfaces. Only when the handshaking of all these signal pairs has happened, the module recognizes that the output data has been transferred, which enable the module to receive new input data. Therefore, the data consistency is guaranteed by this extended handshaking scheme.

2) Switch modules: The switch modules include the modules named *Mux* and *DeMux* as shown in Fig. 1. They are actually the multiplexer and demultiplexer to control the data flow direction, which are 2-to-1 and 1-to-2 respectively. In some stages, the input port of some module needs to switch the data and handshake signals coming from one output port to the another. For example, (4a) does not need the subtraction operation for the calculation of L_{d1}, but (4b) needs it for the calculation of L_{di} when $2 \leq i \leq N$. Similarly, some output port needs to switch its data and handshake signals from one input port to the other. The switching stage number is implemented as a programmable parameter of the switch modules to support static data flow configuration.

3) Memory modules: The memory modules use the on-chip block RAM of the FPGA as the main storage components in the hardware implementation. The objectives of using memory modules in our system are summarized as follows.

- Reuse the data among the algorithm iterations as for the main variables y, v, λ, s.
- Buffer the data feedback between two neighboring subproblems as for $L_{sdi-1}L_{sdi-1}^T$ in (4b).
- Share data access for different computation modules.
- Buffer the data stream between two matrix computation modules to bridge the gap of different throughput rates or data sequences.

The memory modules are all implemented in the same structure with various programmable parameters that determine the memory size, read-out sequence and read times. They are configured according to the data access behaviour of their corresponding variables in the target algorithm to realize distributed data management.

V. Experimental Results

A. Benchmark example

As in [5] and [7], a mass-spring problem is used as benchmark example to evaluate the performance of our implementation. The system consists of p masses connected by springs as illustrated in Fig. 3. The control inputs are the forces $u_1, \ldots, u_p$ which are located at each mass. The control objective is to overcome the horizontal displacement of all masses. There are two states per mass, position and velocity. Both states and control inputs are subject to box constraints. In this MPC problem, the dimensions of the problem are given by:

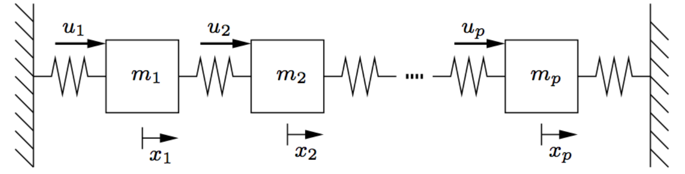

Fig. 3. Mass-spring system.

$$n_x := 2p,\ n_u := p,\ n_y := n_x + n_u,\ n_{ineq} := 2n_y.$$

B. Experimental details

The hardware design is described in VHDL and parameterizable for different MPC problem sizes. It was synthesized, and implemented using Xilinx Vivado 2013.4 design tool targeting a Virtex-7 XC7VX485T. The experiment is built up on a Xilinx VC707 Evalutation Kit, as well as on a desktop PC that has an Intel i7-3770 CPU with a clock frequency of 3.4-3.9 GHz, 16 GB RAM, and is running Windows 7. To communicate with the host PC through the Ethernet interface, our custom design is integrated into a MicroBlaze System-on-Chip test framework as an accelerator. With this interface, we can exchange data between MATLAB running on the PC and the MPC solver implemented on the FPGA board. A closed-loop simulation framework was developed that simulates the mass-spring system in MATLAB and solves the MPC problem using either the FPGA or the C code implementation.

The software implementations of the solver were obtained with the `FORCES` code generator [20] using single precision floating point numbers, and compiled by Microsoft Visual C++ 2010 with compiler optimization set to maximize speed (O2). Both hardware and software implementations are further optimized for the box constraints applied in the benchmark MPC problem. The FPUs in the FPGA implementations are configured to use the maximum number of DSP blocks and the minimum number of clock cycles which support the target clock frequency of the full system.

The correctness of the FPGA implementation was verified by comparing the returned results with the ones obtained from the C code. The differences were in the order of the machine precision which we estimate are a consequence of different data paths in the implementations.

C. Implementation analysis

In all experiments both software and hardware solvers are set to run 10 interior point iterations in each of 500 control cycles. We implemented hardware solvers in the target

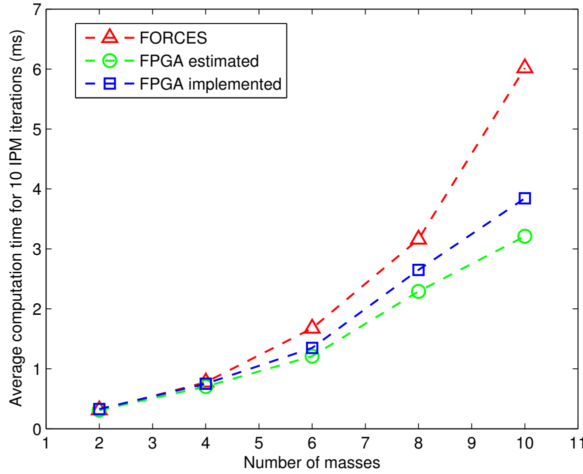

Fig. 4. Average computation times of CPU and FPGA implementations for solving problems with different number of masses and constant prediction horizon $(N = 10)$.

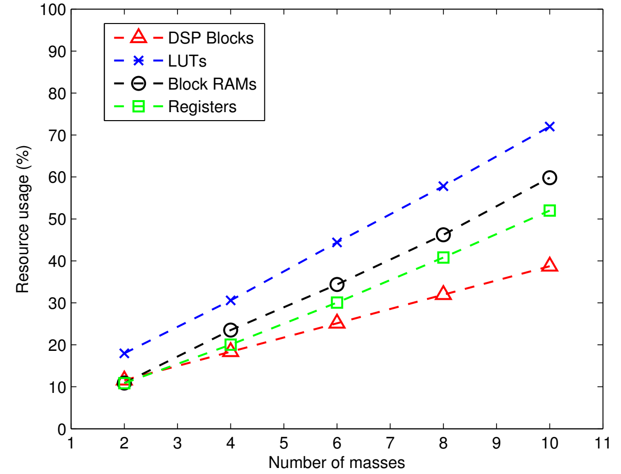

Fig. 5. Resource usage of FPGA implementation for problem instances with varying number of masses and constant horizon $(N = 10)$.

FPGA for problems with 2, 4, 6, 8, and 10 masses. The implementations comprising up to 6 masses are clocked at 250 MHz, the instances with 8 and 10 masses are clocked at 200 MHz. The achievable clock frequency of the FPGA implementation for the larger problem instances decreases with the significantly increased place-and-route complexity, which is caused by the large utilization of DSP and RAM blocks allocated discretely inside FPGA. The power consumption of the implementations including the on-chip testing framework estimated by the Xilinx design tool is in the range from 4 to 6 Watts. This is significantly lower than the power consumption of a high-end desktop CPU and makes the deployment on an embedded control platform feasible.

Fig. 4 shows how the average computation time needed by the FPGA and software implementation is affected when the number of masses, *i.e.*, stage variables, is increased. The horizon N in all instances is equal to 10. The FPGA performance estimation is based on the computational delay and the throughput rate of the systolic-array based pipelines. The delay of the hand-shake protocol and the line search in lines 5 and 9 of Algorithm 1 are not taken into account, which leads to the difference between the implemented and estimated performance of the FPGA implementation. For small problem sizes, the FPGA solver exhibits similar performance as the high-end CPU. With an increasing number of stage variables, however, the hardware implementation is able to outperform its software counterpart. This is due to the increased parallelism in the linear algebra kernels for bigger problem instances.

Fig. 5 shows that the usage of all resources increases linearly with the number of masses. This is a consequence of the column-wise parallelization used in the linear algebra operations and the size of the block matrices which is given by the number of states and inputs. Due to this aggressive parallelism, the LUT resource usage is the bottleneck of the current implementation, which is mostly used for data path control and FPUs.

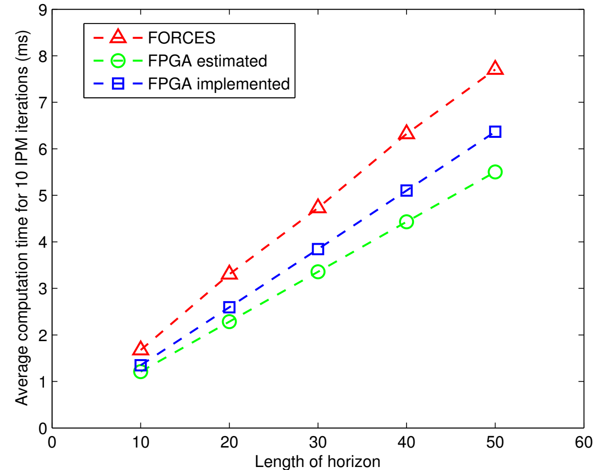

Fig. 6. Performance comparison of CPU and FPGA solving same mass-spring problems with various horizon length and constant 6 masses.

To assess the impact of the horizon length on execution time and resource usage, we implemented another four hardware solvers clocked at 250MHz for the problem instance with 6 masses and horizon lengths 20, 30, 40, and 50. The computational effort of sparse MPC solvers depends linearly on the prediction horizon. This relationship is confirmed for both the hardware and the software implementation by our experiments whose results are shown in Fig. 6. Since resource usage and memory bandwidth of the FPGA implementation only depend on the size of a single stage (determined by n_x, n_u, n_y and n_{ineq}), the total resource usage is hardly affected by the horizon N. As shown in Fig. 7, only the memory demand increases slightly for horizons larger than 20 while the usage of the

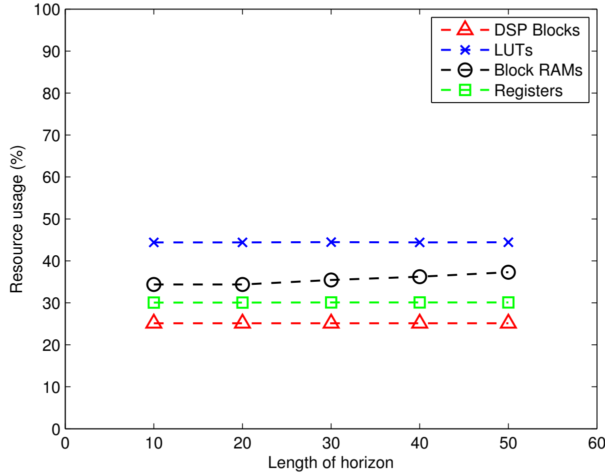

Fig. 7. Resource usage of FPGA implementation for the mass-spring problems with various horizon length and constant 6 masses.

other resources remains constant. Longer horizons just leads to larger memory depths in the hardware implementation, so that more block RAM is used whenever the required memory depth exceeds the depth of the block RAMs. The resource independence from the prediction horizon distinguishes our implementation from the ones described in the literature so far, making it attractive for problems with large horizons.

VI. Conclusion

Computational complexity is one of the main obstacles in the deployment of MPC for applications that require fast sampling times. The paper at hand describes an FPGA implementation of a primal-dual IPM for solving linear quadratic MPC problems fast and efficiently. In contrast to existing hardware solutions, the linear equation system underlying the optimization algorithm is solved using a structure exploiting Cholesky decomposition which allows an efficient implementation of a predictor-corrector scheme, ultimately resulting in a lower number of required IPM iterations. Moreover, the resource usage of our FPGA solver is independent of the prediction horizon MPC making it especially appealing for problems with large horizons. Experiments show a faster execution speed of the FPGA implementation in comparison with an optimized software version running on a high-end desktop CPU with significantly higher power consumption. Its modular design based on the systolic array architecture allows a simple integration of customized computation modules to support special problem classes with minimum resource utilization. We believe that our FPGA-based IPM solver can enable fast MPC applications on future industrial embedded computing platforms with limited CPU performance and/or low power constraints.

Acknowledgment

This work was originally finished as a master thesis project at ABB Corporate Research Center in Switzerland. The authors would like to acknowledge the provision of FPGA test framework from Dr. Andrea Suardi [21], and the support of EPSRC (Grant references EP/I020357/1 and EP/I012036/1).

References

[1] J. M. Maciejowski, *Predictive control: with constraints*. Pearson education, 2002.
[2] 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.
[3] S. Richter, C. Jones, and M. Morari, "Real-time input-constrained MPC using fast gradient methods," in *Decision and Control, 2009 held jointly with the 2009 28th Chinese Control Conference. CDC/CCC 2009. Proceedings of the 48th IEEE Conference on*, 2009, pp. 7387–7393.
[4] Y. Wang and S. Boyd, "Fast model predictive control using online optimization," *Control Systems Technology, IEEE Transactions on*, vol. 18, no. 2, pp. 267–278, 2010.
[5] A. Domahidi, A. Zgraggen, M. Zeilinger, M. Morari, and C. Jones, "Efficient interior point methods for multistage problems arising in receding horizon control," in *IEEE Conference on Decision and Control (CDC)*, Maui, HI, USA, Dec. 2012, pp. 668 – 674.
[6] M. S. Lau, S. Yue, K. Ling, and J. Maciejowski, "A comparison of interior point and active set methods for FPGA implementation of model predictive control," in *Proc. European Control Conference*, 2009, pp. 156–160.
[7] 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*, ser. FPGA '11. New York, NY, USA: ACM, 2011, pp. 209–218.
[8] K.-V. Ling, B. F. Wu, and J. Maciejowski, "Embedded model predictive control (MPC) using a FPGA," in *Proc. 17th IFAC World Congress*, 2008, pp. 15 250–15 255.
[9] A. Wills, A. Mills, and B. Ninness, "FPGA implementation of an interior-point solution for linear model predictive control," in *Preprints of the 18th IFAC World Congress, Milano, Italy*, 2011, pp. 14 527–14 532.
[10] M. A. Boéchat, J. Liu, H. Peyrl, A. Zanarini, and T. Besselmann, "An architecture for solving quadratic programs with the fast gradient method on a Field Programmable Gate Array," in *Proc. 21st Mediterranean Conference on Control and Automation*, Crete, Greece, 2013.
[11] J. Jerez, P. Goulart, S. Richter, G. Constantinides, E. Kerrigan, and M. Morari, "Embedded predictive control on an FPGA using the fast gradient method," in *Control Conference (ECC), 2013 European*, July 2013, pp. 3614–3620.
[12] C. V. Rao, S. J. Wright, and J. B. Rawlings, "Application of interior-point methods to model predictive control," *Journal of optimization theory and applications*, vol. 99, no. 3, pp. 723–757, 1998.
[13] M. Vukov, A. Domahidi, H. J. Ferreau, M. Morari, and M. Diehl, "Auto-generated algorithms for nonlinear model predictive control on long and on short horizons," in *Proceedings of the 52nd Conference on Decision and Control (CDC)*, 2013.
[14] S. Mehrotra, "On the implementation of a primal-dual interior point method," *SIAM Journal on Optimization*, vol. 2, no. 4, pp. 575–601, 1992.
[15] S. J. Wright, *Primal-dual interior-point methods*. SIAM, 1997, vol. 54.
[16] "Xilinx AXI Reference Guide," 2012. [Online]. Available: http://www.xilinx.com/support/documentation/ip_documentation/axi_ref_guide/v14_1/ug761_axi_reference_guide.pdf
[17] H. Kung, "Why systolic architectures?" *IEEE Computer*, vol. 15, no. 1, pp. 37–46, 1982.
[18] "AMBA 4 AXI4-Stream Protocol." [Online]. Available: http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.set.amba/index.html
[19] "Xilinx LogiCORE IP Floating-Point Operator v6.0 Data Sheet," 2012. [Online]. Available: http://www.xilinx.com/support/documentation/ip_documentation/floating_point/v6_0/ds816_floating_point.pdf
[20] A. Domahidi, "FORCES: Fast optimization for real-time control on embedded systems," http://forces.ethz.ch, Oct. 2012.
[21] A. Suardi, "ICL SDK4FPGA." [Online]. Available: https://github.com/asuardi/SDK4FPGA