Hardware Tailored Linear Algebra for Implicit Integrators in Embedded NMPC.

Frison, Gianluca; Quirynen, Rien; Zanelli, Andrea; Diehl, Moritz; Jørgensen, John Bagterp

Published in:
IFAC-PapersOnLine

Link to article, DOI:
10.1016/j.ifacol.2017.08.2026

Publication date:
2017

Document Version
Publisher's PDF, also known as Version of record

Available online at www.sciencedirect.com

ScienceDirect

IFAC Papers Online CONFERENCE PAPER ARCHIVE

IFAC PapersOnLine 50-1 (2017) 14392–14398

Hardware Tailored Linear Algebra for Implicit Integrators in Embedded NMPC ⋆

Gianluca Frison [*,**] **Rien Quirynen** [**] **Andrea Zanelli** [**]
Moritz Diehl [**] **John Bagterp Jørgensen** [*]

[*] *Department of Applied Mathematics and Computer Science, Technical University of Denmark, email: {giaf, jbjo} at dtu.dk*
[**] *Department of Microsystems Engineering, University of Freiburg, email: {gianluca.frison, andrea.zanelli, moritz.diehl} at imtek.uni-freiburg.de, rien.quirynen at esat.kuleuven.be*

Abstract: Nonlinear Model Predictive Control (NMPC) requires the efficient treatment of the dynamic model in the form of a system of continuous-time differential equations. Newton-type optimization relies on a numerical simulation method in addition to the propagation of first or higher order derivatives. In the case of stiff or implicitly defined dynamics, implicit integration schemes are typically preferred. This paper proposes a tailored implementation of the necessary linear algebra routines (LU factorization and triangular solutions), in order to allow for a considerable computational speedup of such integrators. In particular, the open-source BLASFEO framework is presented as a library of efficient linear algebra routines for small to medium-scale embedded optimization applications. Its performance is illustrated on the nonlinear optimal control example of a chain of masses. The proposed library allows for considerable speedups and it is found to be overall competitive with both a code-generated solver and a high-performance BLAS implementation.

Keywords: Nonlinear predictive control, Embedded optimization, Computer software

1. INTRODUCTION

Optimization based control and estimation techniques have gained an increasing popularity, because of their ability to directly treat the, possibly nonlinear, constraints, system dynamics and objectives. This paper targets the use of Nonlinear Model Predictive Control (NMPC) (Mayne and Rawlings, 2013), which requires the solution of a nonlinear nonconvex Optimal Control Problem (OCP) at each sampling instant. Especially in case of fast dynamic systems, the corresponding computational burden can form a major challenge. Note that Moving Horizon Estimation (MHE) techniques typically require the online solution of a similar OCP formulation. For this purpose, tailored online algorithms have been developed for real-time optimal control as discussed in (Diehl et al., 2009; Kirches et al., 2010; Ohtsuka, 2004).

An important algorithmic ingredient in any direct optimal control method consists of the integration scheme for the numerical simulation of the nonlinear system of differential equations (Bock and Plitt, 1984). Especially in case of stiff or implicitly defined dynamics, implicit integrators are typically used to efficiently provide an accurate discretization of the continuous-time problem (Hairer and Wanner, 1991). When using a Newton-type optimization algorithm (Nocedal and Wright, 2006), one additionally needs the efficient propagation of first or possibly even second order derivatives (Griewank and Walther, 2008; Quirynen et al., 2017).

As discussed in (Quirynen et al., 2015, 2016), the computational bottleneck in the implementation of such implicit integrators with tailored sensitivity propagation, typically corresponds to the Linear Algebra (LA) routines. Note that, in combination with structure exploiting optimal control algorithms, dense instead of sparse LA packages can be used in general (Frison, 2015). More specifically, the present paper focuses on the efficient factorization of the Jacobian matrix and its reuse for the corresponding linear system solutions in the implicit integrator. For this purpose, high-performance open-source BLAS implementations can be used such as GotoBLAS (Goto and Geijn, 2008), OpenBLAS (OpenBLAS, 2011) and BLIS (Van Zee and van de Geijn, 2015), as well as proprietary implementations such as Intel's MKL and AMD's ACML. In the case of small embedded applications, the technique of automatic code generation has been shown to be relatively successful (Houska et al., 2011; Mattingley et al., 2010).

Unlike the above mentioned BLAS implementations (which are generally optimized for large-scale problems) and code generation tools (targeting small-scale systems), we propose highly efficient implementations for a wide range of small to medium-scale embedded optimization applications. The open-source BLASFEO (Frison et al., 2017) framework is introduced which provides highly optimized dense LA routines, specially tailored to give the best performance for the typical matrix sizes of interest. Cur-

⋆ This research was supported by the EU via ERC-HIGHWIND (259 166), ITN-TEMPO (607 957), ITN-AWESCO (642 682), by the DFG in context of the Research Unit FOR 2401, and by Det Frie Forskningsråd (DFF - 6111-00398).

2405-8963
Peer review under responsibility of International Federation of Automatic Control.
10.1016/j.ifacol.2017.08.2026

rently, BLASFEO provides three implementations: a high-performance one (further developing the implementation scheme originally proposed in HPMPC (Frison et al., 2014)), a reference one (coded in plain C with focus on small-scale performance) and wrappers to key BLAS and LAPACK routines. The high-performance implementation is the most noteworthy. It provides a matrix format optimizing cache usage for small to medium size matrices, and assembly-coded LA kernels optimized to exploit the specific hardware features of many computer architectures. Therefore, BLASFEO provides efficient LA routines for a wide range of matrix sizes.

The paper is organized as follows. Section 2 introduces the use of implicit integrators with a direct sensitivity propagation for embedded predictive control. Section 3 discusses the efficient implementation of the corresponding LA routines, with a focus on the LU factorization and Section 4 presents the open-source implementation in the BLASFEO framework. Section 5 illustrates the numerical performance of the proposed implementation, including the case study of the nonlinear optimal control of a chain of masses. Section 6 finally concludes the paper.

Contribution: this paper presents the efficient implementation of the dense LU factorization and relative triangular solutions in the BLASFEO framework, tailored to be used in implicit integrators for embedded optimization applications. Even though the proposed approach follows the guidelines of the HPMPC library, using register blocking, vectorization based instruction sets and a tailored panel-major storage format, this paper features a novel hardware tailored implementation of the LU factorization, including the efficient implementation of partial pivoting. The contribution of the present paper is orthogonal to the improvements in integration algorithms, as it can be combined with any implicit integration scheme. Depending on the problem size, a speedup of up to one order of magnitude is observed, with respect to code-generated LA currently employed in embedded optimization.

2. IMPLICIT INTEGRATORS FOR EMBEDDED PREDICTIVE CONTROL

Let us consider the following parametric optimal control problem (OCP) formulation in discrete time

$$\begin{aligned}
\min_{X,\,U} \quad & \sum_{i=0}^{N-1} l(x_i,u_i) + m(x_N) && \text{(1a)}\\
\text{s.t.} \quad & 0 = x_0 - \hat{x}_0, && \text{(1b)}\\
& 0 = x_{i+1} - \phi(x_i,u_i), \quad i=0,\dots,N-1, && \text{(1c)}\\
& 0 \geq h(x_i,u_i), \qquad\quad i=0,\dots,N-1, && \text{(1d)}\\
& 0 \geq r(x_N), && \text{(1e)}
\end{aligned}$$

with state trajectory $X = [x_0^\top,\dots,x_N^\top]^\top$, where $x_i \in \mathrm{R}^{n_x}$ and control inputs $U = [u_0^\top,\dots,u_{N-1}^\top]^\top$, where $u_i \in \mathrm{R}^{n_u}$. The objective (1a) consists of a stage cost $l(\cdot)$ for each of the intervals $i = 0,\dots,N-1$, and a terminal cost $m(\cdot)$. The function $\phi(\cdot)$ defines the continuity constraint in (1c). Path constraints can be specified in (1d), in addition to a terminal constraint (1e). The parametric optimization problem depends on $\hat{x}_0$ in (1b), which denotes the current state estimate of the system.

Note that the structured OCP in Eq. (1) is common in practice and can, for example, be obtained using direct multiple shooting for a continuous-time OCP formulation as in (Bock and Plitt, 1984).

2.1 Online Algorithms for Nonlinear MPC

Any nonlinear optimization solver can be used to solve the NLP in Eq. (1). This paper considers a Sequential Quadratic Programming (SQP) approach, which has been shown to work well for Nonlinear MPC in (Diehl et al., 2009). It solves a sequence of Quadratic Program (QP) approximations and converges to a locally optimal solution of the original NLP (Nocedal and Wright, 2006). When the OCP consists of a least-squares type objective, the Generalized Gauss-Newton (GGN) method (Bock, 1983) is typically used. All nonlinear functions in the NLP need to be linearized at each SQP iteration. For this purpose, Algorithmic Differentiation (AD) (Griewank and Walther, 2008) can be used to efficiently evaluate derivatives.

In NMPC, one needs to solve this parametric NLP (1) at each time step. An efficient continuation technique is part of the Real-Time Iteration (RTI) scheme as presented in (Diehl et al., 2002). The aim is to minimize the computational delay between obtaining the new state estimate $\hat{x}_0$ and applying the next control input. There are different options to efficiently solve the large structured QP subproblem. For our case study in Section 5, the combination of condensing with the embedded active-set solver qpOASES is used (Ferreau et al., 2014).

2.2 Implicit Integrators in Newton-type Optimization

The dynamic model is typically described as

$$0 = f(\dot{x}(t), x(t), u(t)), \tag{2}$$

which denotes a set of implicit Ordinary Differential Equations (ODE), where the Jacobian $\frac{\partial f}{\partial \dot{x}}(\cdot)$ is assumed to be invertible. The function $\phi(x_i,u_i)$ in the continuity constraint (1c) then represents a numerical simulation of these dynamics (2) over the interval $[t_i, t_{i+1}]$, starting from the initial value $x(t_i) = x_i$ and applying the input u_i. Note that this can be readily extended to an index-1 system of Differential-Algebraic Equations (DAE).

For systems having stiff or implicit dynamics, implicit integration schemes are typically preferred as they allow for performing these numerical simulations at a lower computational cost than explicit schemes (Hairer and Wanner, 1991). Let us consider the popular class of Implicit Runge-Kutta (IRK) schemes, for which one integration step can be written in a compact form as

$$\begin{aligned}
x_{i,n+1} &= F(x_{i,n}, K_{i,n}, u_i)\\
0 &= G(x_{i,n}, K_{i,n}, u_i),
\end{aligned} \tag{3}$$

for $n = 0,\dots,N_\mathrm{s}-1$ where N_s denotes the number of integration steps. The simulation result x_{i,N_s} can then be used to define the function $\phi(x_i,u_i)$, given the initial state value $x_{i,0} = x_i$. In the case of an s-stage IRK formula, the auxiliary variables $K_{i,n} \in \mathrm{R}^{s\,n_x}$ are defined for each integration step.

For a well-defined set of dynamics (2), the Jacobian $\frac{\partial G}{\partial K}(\cdot)$ is invertible (Hairer and Wanner, 1991). This means that the stage variables $K_{i,n}$ can be computed iteratively

$$K_{i,n}^{[j+1]} = K_{i,n}^{[j]} - \frac{\partial G}{\partial K}^{-1} G(x_{i,n}, K_{i,n}^{[j]}, u_i), \qquad (4)$$

where j denotes the iteration number, using a Newton-type implementation. Note that one can avoid performing multiple inner iterations, by lifting such an implicit integration scheme in a Newton-type optimization algorithm as discussed in (Quirynen et al., 2015, 2016). Based on the implicit function theorem, also the first order forward sensitivities can be computed as

$$\frac{\mathrm{d}K_{i,n}}{\mathrm{d}(x_i, u_i)} = -\frac{\partial G}{\partial K}^{-1} \left(\frac{\partial G}{\partial x_{i,n}} \frac{\mathrm{d}x_{i,n}}{\mathrm{d}(x_i, u_i)} + \begin{bmatrix} 0 & \frac{\partial G}{\partial u_i} \end{bmatrix} \right). \quad (5)$$

Note that both the numerical elimination of the IRK variables in (4) and the propagation of first order sensitivities (5), requires the efficient computation of a factorization for the matrix $\frac{\partial G}{\partial K}$ and the corresponding linear system solutions. This paper focuses on these algorithmic aspects and presents a hardware tailored software implementation for the associated LA routines.

3. EFFICIENT LINEAR SYSTEM SOLUTION

The matrix $\frac{\partial G}{\partial K}$ in (4) and in (5) is in general not symmetric and, in this paper, it is considered as a dense matrix. It can therefore be factorized using a dense LU factorization.

3.1 Introduction to the LU Factorization

The LU factorization can be employed to factorize a generic $n \times n$ matrix A as the product of a lower triangular matrix L and an upper triangular matrix U, at a computational cost of about $\frac{2}{3}n^3$ flops (Golub and Loan, 1996). The basic algorithm is not numerically stable, and it can break down if, at some point, a zero is found on the diagonal. The stability of the algorithm can be improved by properly reordering rows and/or columns of A. This operation is commonly referred to as pivoting. A recent survey of pivoting techniques for the LU factorization can be found in (Donfack et al., 2015).

The most common pivoting technique is known as *partial pivoting*, where only row swaps are performed. This is equivalent to finding the LU factorization of the matrix PA, where P is a permutation matrix. It can be proved that an LU factorization with partial pivoting exists for any square matrix (Golub and Loan, 1996). While, in theory, the LU factorization with partial pivoting can be numerically unstable, decades of experience has shown that, in practice, it can be considered stable.

Partial pivoting has a computational complexity of $\mathcal{O}(n^2)$ comparisons such that, asymptotically, its cost is hidden by the $\mathcal{O}(n^3)$ flops required by the LU factorization. However, it can heavily affect the performance due to the intrinsic non-locality of the search, making the use of algorithm-by-blocks difficult. It can prevent the use of efficient algorithm-by-blocks strategies and it complicates the parallelization of the algorithm. In the worst case, the row swaps have also a complexity of $\mathcal{O}(n^2)$ memory copies, and it can be more computationally expensive than the search for the pivot element.

Remark 1. (Warm-started pivoting). While it is generally not possible to avoid the search for the pivot element (unless diagonally dominant matrices are considered), it can be possible to reduce the overall amount of row swaps in the context of real-time optimal control. As discussed in the previous section, each iteration of the Newton-type optimization algorithm requires the numerical simulation of the dynamic system. In order to reduce the overall amount of row swaps for the LU factorization and the linear system solutions in each integration step, one could therefore prematurely apply a permutation to the linear system, based on the results of the previous integration step. We further refer to this as *warm-started pivoting*.

3.2 Efficient Matrix Storage Format

One of the key features of the implementation in HPMPC, on which the developments in BLASFEO are based, is the use of a special matrix storage format, referred to as panel-major in (Frison, 2015). In this format, each matrix is divided into panels, which are defined as sub-matrices with (many) more columns than rows. In BLASFEO, each panel has a fixed and small number of rows (generally 2, 4 or 8), that is chosen depending on the computer architecture. The panels are stored, one after the other, in memory in a row-block-major format. Within each panel, the matrix elements are stored in column-major format.

For matrices fitting in the last level of cache, this matrix format is (nearly) the optimal storage for level 3 BLAS routines. In fact, it roughly corresponds to the first level of packing employed in high-performance implementations of BLAS such as GotoBLAS (Goto and Geijn, 2008), OpenBLAS (OpenBLAS, 2011) and BLIS (Van Zee and van de Geijn, 2015). In high-performance BLAS implementations, the default matrix format is column-major. However, internally, sub-matrices of the left factor A and of the right factor B are copied into buffers such that matrix elements are stored in the exact same order as they are streamed by the `gemm`, i.e., the general matrix-matrix multiplication kernel (allowing for efficient data prefetch and optimal cache usage). By carefully choosing buffer sizes and the operation order, it is possible to ensure that these buffers are held in the L1 and L2 cache, respectively. This allows the LA kernels to achieve near maximum throughput when operating on data held in the buffers. The main drawback of this approach is that the copying of data into the buffers is performed online, and it can heavily affect performance in case of small to medium size matrices.

Figure 1 shows an example of the product of two matrices stored using the panel-major format. More precisely, the performed operation is the 'NT' variant of `gemm`, which is the optimal variant for matrices stored in the panel-major format (Frison, 2015). In this example, the kernel computes a 4×2 sub-matrix of the result. The panel height b_s is the same for the left and for the right operands, as well as for the result matrix (and equal to 2 in this example, since each square sub-matrix of the result matrix has 2 columns of 2 elements each). Conversely, the buffers employed internally in optimized BLAS implementations can have a different value of b_s for the left and the right factors (optimally chosen equal to the number of rows swept by the kernel, in this case 4 and 2 for A and B respectively), while the result matrix would be directly stored in column-major format.

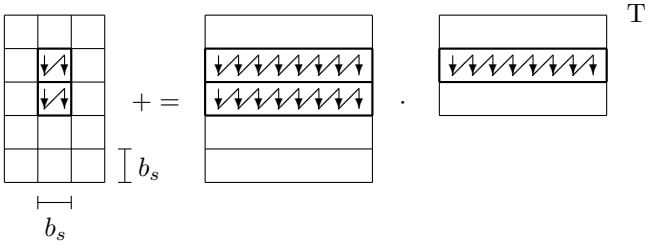

Fig. 1. Panel-major matrix storage format: elements are stored in the same order as the `gemm` kernel accesses them. This `gemm` kernel implements the optimal 'NT' variant (namely, A non-transposed, B transposed). The panel height b_s is the same for the left and the right operand, as well as for the result matrix.

By fixing b_s for all matrices, it is possible to use the panel-major format as a generic matrix storage format, since it can be used for any argument in the LA routines. In an embedded optimization framework, this allows for the conversion from standard matrix formats into panel-major to be performed off-line, or outside the optimization algorithm loop (and therefore being well amortized over the algorithm iterations also in case of small matrices, unlike other BLAS implementations).

3.3 High-performance Small-scale LU Factorization

The LU factorization with partial pivoting is implemented in the LAPACK routines `getrf` (blocked algorithm) and `dgetf2` (unblocked algorithm). The unblocked algorithms in LAPACK are implemented using level 2 BLAS, and are aimed at small matrices. Conversely, blocked algorithms in LAPACK are implemented using level 3 BLAS and unblocked LAPACK algorithms, and are aimed at large matrices. The block size is generally chosen such that working sub-matrices can fit in some cache level (generally L1, giving a typical block size of 32 or 64). Cache blocking has the aim of improving the overall performance for relatively large matrices. The notion of blocked algorithms and BLAS levels can be found in (Golub and Loan, 1996).

The main drawback of this approach is that, for matrices smaller than the block size, the LAPACK routines are implemented using solely level 2 BLAS. Therefore, there is no reuse of matrix elements, once they are moved into the registers. This strongly affects the performance in case of small matrices, which occur typically in embedded optimization applications.

In the BLASFEO framework, we implement the LU factorization routine (and more general LAPACK routines) as if it was a level 3 BLAS routine, and not on top of it. This means that tailored factorization kernels are implemented, operating on sub-matrices of the result, fitting in the registers of the processor. This register blocking provides reuse of matrix elements for much smaller matrices than in the case of cache blocking, greatly enhancing small-scale performance (Frison, 2015).

In BLASFEO, the order of the two outer loops around the LA kernels can affect the performance scalability with the matrix size. Since there are generally more rows than columns in the LA kernels, the left `gemm` factor (in what follows, A) is streamed in larger sub-matrices than the right `gemm` factor (B). Therefore, better performance can be achieved if the sub-matrix of A is kept in L1 cache between subsequent calls to the LA kernel, while the panels from B are streamed from L2 cache. This can be achieved by properly choosing the order of the two outer loops. In particular, by having the outermost loop over the rows of the result matrix and the middle loop over the columns, such that the result is computed in row-blocks.

In case of the LU factorization, the search for the pivot element prevents the use of the optimal loop order, since the search is performed over the entire column below the diagonal element. This also prevents the strict use of algorithm-by-blocks for the computation of L: each matrix element needs to be computed in two steps, one to compute the pivot, and one to scale the column below the diagonal once the pivot has been found. However, the computation of the upper factor U can be entirely performed using algorithm-by-blocks and optimal loop ordering.

As a final note, the kernels in the LU factorization are based on the 'NN' (both A and B non-transposed) variant of the `gemm` routine. In the BLASFEO framework, this generally gives slightly worse performance than the optimal 'NT' variant (A non-transposed, B transposed), since the matrix B is streamed in a non-contiguous way.

3.4 Triangular System Solutions

In this paper, the LU factorization is employed in the solution of systems of linear equations

$$PAX = LUX = PB, \tag{6}$$

where P is a permutation matrix and the matrix A is not assumed to be symmetric. In general, X and B can be matrices, i.e., in the multiple right-hand side (RHS) case. Note that a multiplication on the left by a permutation matrix gives a permutation of the rows. The solution of triangular systems of linear equations with the lower triangular matrix L (forwardsolve) and the upper triangular matrix U (backsolve) in (6) is implemented as

$$X = U^{-1}(L^{-1}(PB)).$$

In the above solution scheme (referred to as 'N' scheme), the triangular matrix is always on the left. Therefore, in the LA kernel, a large triangular matrix is used to solve a system with a smaller number of RHSs, resulting in high dependency between instructions. And both operands are non-transposed, such that the solve routines are based on the sub-optimal 'NN' variant of `gemm`.

A better implementation is obtained by transposing both sides in (6), as

$$X^T U^T L^T = (PAX)^T = (PB)^T = B^T P^T. \tag{7}$$

Note that a multiplication on the right by a permutation matrix gives a permutation of the columns. The forwardsolve and backsolve in (7) are implemented as

$$X^T = ((B^T P^T)L^{-T})U^{-T}.$$

In the above solution scheme (referred to as 'T' scheme), the triangular matrix is always on the right. Therefore, in the LA kernel, a small triangular matrix is used to solve a system with a larger number of RHSs, resulting in many independent instructions. The left operand is non-

transposed while the right one is transposed, such that the routines are based on the optimal 'NT' variant of `gemm`.

The 'T' scheme requires the transposed B^T of the RHS and returns the transposed X^T of the result matrix. If the transposition is explicitly performed, this forms an overhead. In some cases, it is possible to avoid that, e.g., if the transposition is embedded in the conversion of the data matrices between column- and panel-major storage formats, and therefore performed at no extra cost.

4. OPEN-SOURCE BLASFEO FRAMEWORK

The LA routines, employed in this paper, are implemented in the recently developed, open-source BLASFEO framework (Frison et al., 2017). This is a further development of the LA framework originally proposed in HPMPC (Frison et al., 2014), a library for the High-Performance implementation of solvers for MPC.

Compared to the LA routines in HPMPC, BLASFEO introduces a novel interface, that defines matrix (`strmat`) and vector (`strvec`) structure types to completely hide any implementation detail about the storage format. Three implementations are provided: a high-performance version (HP, providing assembly-coded LA kernels optimized for many computer architectures and employing the panel-major matrix format), a reference version (RF, written in plain C code with 2×2 register block size and employing column-major matrix format) and a wrapper to key BLAS and LAPACK routines (WR, employing column-major matrix format). Furthermore, the HP and RF implementations store an extra vector that can be employed to hold, e.g., the inverse of the diagonal (computed anyway in the factorization process), eliminating the need to re-compute expensive divisions in the solve routines. The three implementations together allow one to code an algorithm once, while choosing the best LA option, depending on the specific problem dimensions.

Conversion and cast routines are provided to manage the coexistence of `strmat` and standard matrix formats in the same piece of code. In the particular case of the RF implementation, there is no need to convert between column-major and `strmat`. Therefore, the reference version is generally the best choice for small matrices (matching or exceeding the performance of code-generated triple-loop based LA routines). The one-time conversion cost is generally well amortized for medium size matrices, where the HP version performs better. For very large matrices, the WR version may be the best choice since generally BLAS is well optimized for the large-scale case and it comes with a multi-thread implementation.

5. NUMERICAL EXPERIMENTS

In this section, numerical results are reported that show the benefits of the proposed implementations. All numerical simulations are carried out on a modern computer architecture (Intel Core i7 4800MQ processor). The processor architecture is Haswell, supporting the AVX2 and FMA3 instruction sets (ISAs). Each core can execute two 256-bit wide fused-multiply-add instructions every clock cycle. At 3.3 GHz (maximum frequency when the 256-bit wide execution units are active), this gives a maximum

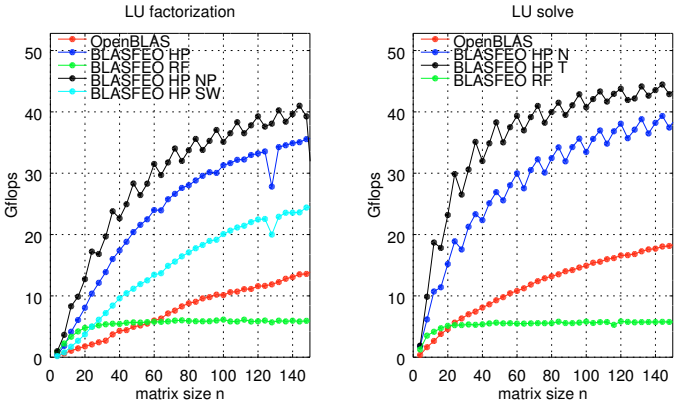

Fig. 2. Performance plot for LU factorization (left) and relative solves (right) for square $n \times n$ matrices.

throughput per core of 52.8 Gflops (billions of floating-point operations per second) in double precision. The maximum frequency when the 256-bit wide execution units are not active is 3.7 GHz. Only single-threaded code is considered. The operating system is a 64-bit Ubuntu distribution with Linux kernel 4.4, the compiler is gcc 5.4.0.

5.1 Computational Performance

Figure 2 shows performance plots for the LU factorization and relative forward and backward solves: different implementations of the same LA routine are compared, and the fastest implementation can perform the same amount of work in shorter time, therefore resulting in higher computational performance (in Gflops). The x-axis is the matrix size n (up to 150, which is enough to cover the largest test in Section 5.2), while the y-axis is the computational performance in Gflops (the maximum throughput is at 52.8 at the top of the figures). No conversions between matrix formats are performed.

For the LU factorization with pivot (left), the high-performance (HP, blue) and reference (RF, green) versions of BLASFEO are compared to OpenBLAS (red) in case no row swaps occur. BLASFEO HP gives very good performance for all considered matrix sizes. BLASFEO RF performs well for small matrices, but then the performance flats out at relatively low levels since the generic C code does not explicitly target powerful hardware ISAs. OpenBLAS gives much lower performance for small matrices. The BLASFEO HP version is also tested in the worst case, where n row swaps occur (SW, cyan): the performance penalty is considerable. Additionally, an HP implementation of the LU factorization without pivoting (NP, black) is also plotted as a reference, to show the performance penalty due to the search for the optimal pivot. From the picture it is clear that the performance penalty of performing row swaps may be quite larger than the performance penalty of looking for the pivot: this motivates the use of warm-started pivoting in Remark 1.

For the LU solve (right), the 'N' solution scheme is implemented using BLASFEO HP (blue), BLASFEO RF (green) and OpenBLAS (red). The findings are similar to the LU factorization case. Furthermore, a BLASFEO HP implementation of the 'T' solution scheme is considered (black), which gives a noticeable extra speedup.

5.2 NMPC Case Study: Chain of Masses

This section illustrates the performance of the implicit integrators for embedded optimization, based on the proposed LA routines in the BLASFEO framework, for the example of a Nonlinear MPC case study. As discussed in Section 2.1, a C-code implementation of the Gauss-Newton based RTI scheme is used in combination with a lifted collocation integrator (Quirynen et al., 2016) and condensing with qpOASES. The evaluation of derivatives is performed using Algorithmic Differentiation (AD) in the open-source CasADi framework (Andersson et al., 2012).

We consider the chain mass control problem, presented in (Wirsching et al., 2006). The task of the controller is to return a chain of masses connected with springs to its steady state, starting from a perturbed initial configuration. In this paper, the setup consists of n_{m} masses where this number can be changed to scale the dimensions of the problem. The mass at one end is fixed, while the acceleration vector of the mass at the other end is the system input $u \in \mathrm{R}^3$. The state of each free mass consists in its position and velocity, such that the system can be described by the state vector $x \in \mathrm{R}^{6(n_{\mathrm{m}}-1)}$. An NMPC scheme with a horizon length of $T = 2$ s and $N = 20$ equidistant intervals with a sampling time of $T_{\mathrm{s}} = 0.1$ s is used. The OCP formulation is subject to simple bounds on the control inputs $u_i \in [-10, 10],\ i = 1, 2, 3$ and to the system dynamics, which are described by an explicit set of Ordinary Differential Equations (ODE). Within each shooting interval, a fixed number of N_{s} integration steps of an s-stage Gauss collocation scheme is used.

Table 1 shows the average computation times for the different algorithmic components in one RTI iteration. It can be observed that the implicit integrator, and especially the linear system solution based on an LU factorization, requires a considerable amount of computational effort, e.g., in case of the code-generated (CG) triple-loop LA. The CG implementation fixes the size of all loops at code generation time and the compiler can exploit this additional information, e.g., to unroll loops. If the BLASFEO HP version is employed, the total simulation time and the QP solution time are about the same, for this example. It is interesting to notice that the AD code and the QP solution times are not the same for different LA versions, even if they do not make use of the LA libraries (and therefore the exact same code is executed in all cases). This is due to the different frequency the processor can run at: BLASFEO HP or OpenBLAS employ the 256-bit wide execution units, resulting in about 10% lower processor frequency (3.3 GHz vs 3.7 GHz). This is observed clearly in this test, but applies to all performed tests.

Table 2 reports in more detail the timings for the LA code, in case $s = 1$ or 3 collocation stages are employed. The number of free masses is varied between 1 and 8, resulting in a number of states n_x ranging between 6 and 48. This results in a minimum size of the matrix $\frac{\partial G}{\partial K}$ equal to 6, a maximum equal to 144. Similarly, the number of columns in the right-hand side ranges between 10 and 52. BLASFEO HP, BLASFEO RF, CG triple-loop and OpenBLAS are compared, when LU factorization with partial pivoting and the 'N' solution scheme are employed. Furthermore, a BLASFEO HP based implementation of

Table 1. Detailed timing results (in ms) for Gauss-Newton based RTI on the chain mass control example ($N = 20$, $N_{\mathrm{s}} = 2$, $s = 2$).

$N_{\mathrm{s}} = 2$, $s = 2$	BLASFEO		triple-loop	OpenBLAS
$n_{\mathrm{m}} = 3$, $n_x = 18$	HP	RF	CG	
AD code	0.058	0.052	0.051	0.058
LA routines	**0.222**	**0.643**	**1.216**	**0.699**
Total simulation	0.435	0.828	1.405	0.908
QP solution	0.587	0.532	0.534	0.584
$n_{\mathrm{m}} = 5$, $n_x = 30$				
AD code	0.170	0.145	0.147	0.181
LA routines	**0.731**	**2.683**	**4.900**	**2.064**
Total simulation	1.310	3.190	5.413	2.648
QP solution	1.635	1.300	1.393	1.528
$n_{\mathrm{m}} = 7$, $n_x = 42$				
AD code	0.397	0.342	0.339	0.382
LA routines	**1.680**	**7.083**	**12.976**	**4.036**
Total simulation	2.965	8.170	14.076	5.257
QP solution	2.689	2.544	2.542	2.667

LU factorization with partial pivoting and the 'T' solution scheme is considered. For the smallest problem, CG triple-loop gives the best performance, but it is closely followed by BLASFEO RF. For all but the smallest problem, BLASFEO HP gives the best performance, in particular if the 'T' solution scheme is employed. Compared to CG triple-loop LA routines, generally employed in embedded optimization, the LA provided by BLASFEO HP can give a speedup of up to one order of magnitude, in case of large problems and many collocation stages.

6. CONCLUSIONS AND OUTLOOK

This paper presents a novel implementation framework for the LA routines employed in implicit integration schemes for embedded optimization. The use of hardware-tailored routines can give a speedup of up to an order of magnitude, compared to the code-generated triple-loop LA routines generally employed in embedded applications. The open-source BLASFEO framework provides an high-performance implementation of the LU factorization and triangular solution routines, optimized for many computer architectures. The LA in BLASFEO can be combined with any numerical algorithm, requiring dense LA routines, to provide efficient solutions for a wide class of applications.

REFERENCES

Andersson, J., Akesson, J., and Diehl, M. (2012). CasADi – a symbolic package for automatic differentiation and optimal control. In *Recent Advances in Algorithmic Differentiation*, volume 87 of *Lecture Notes in Computational Science and Engineering*, 297–307. Springer.

Bock, H.G. (1983). Recent advances in parameter identification techniques for ODE. In *Numerical Treatment of Inverse Problems in Differential and Integral Equations*, 95–121. Birkhäuser.

Bock, H.G. and Plitt, K.J. (1984). A multiple shooting algorithm for direct solution of optimal control problems. In *Proceedings of the IFAC World Congress*, 242–247. Pergamon Press.

Table 2. Detailed timings (in ms) for LU factorization and linear system solution for several LA implementations. The number of flops is $\frac{2}{3}m^3 + 2m^2n$.

collocation stages	n_x	m	n	BLASFEO HP N	BLASFEO HP T	BLASFEO RF N	triple-loop CG N	OpenBLAS N
1	6	6	10	0.013	0.011	0.008	0.007	0.028
1	12	12	16	0.021	0.020	0.030	0.031	0.144
1	18	18	22	0.054	0.046	0.078	0.149	0.313
1	24	24	28	0.086	0.085	0.172	0.318	0.446
1	30	30	34	0.140	0.119	0.296	0.564	0.521
1	36	36	40	0.206	0.201	0.530	0.964	0.864
1	42	42	46	0.320	0.279	0.815	1.482	0.936
1	48	48	52	0.393	0.376	1.221	2.146	1.139
3	6	18	10	0.036	0.032	0.048	0.062	0.222
3	12	36	16	0.095	0.099	0.276	0.363	0.625
3	18	54	22	0.271	0.252	0.876	1.555	1.032
3	24	72	28	0.462	0.468	1.983	3.397	1.514
3	30	90	34	0.873	0.816	3.793	6.305	2.090
3	36	108	40	1.275	1.291	6.492	10.885	2.795
3	42	126	46	2.235	2.146	10.130	16.461	3.754
3	48	144	52	2.820	2.813	14.484	28.347	4.781

Diehl, M., Bock, H.G., Schlöder, J., Findeisen, R., Nagy, Z., and Allgöwer, F. (2002). Real-time optimization and nonlinear model predictive control of processes governed by differential-algebraic equations. *Journal of Process Control*, 12(4), 577–585.

Diehl, M., Ferreau, H.J., and Haverbeke, N. (2009). Efficient numerical methods for nonlinear MPC and moving horizon estimation. In L. Magni, M. Raimondo, and F. Allgöwer (eds.), *Nonlinear model predictive control*, volume 384 of *Lecture Notes in Control and Information Sciences*, 391–417. Springer.

Donfack, S., Dongarra, J., Faverge, M., Gates, M., Kurzak, J., Luszczek, P., and Yamazaki, I. (2015). A survey of recent developments in parallel implementations of gaussian elimination. *Concurr. Comput. : Pract. Exper.*, 27(5), 1292–1309.

Ferreau, H.J., Kirches, C., Potschka, A., Bock, H.G., and Diehl, M. (2014). qpOASES: a parametric active-set algorithm for quadratic programming. *Mathematical Programming Computation*, 6(4), 327–363.

Frison, G., Sorensen, H.B., Dammann, B., and Jørgensen, J.B. (2014). High-performance small-scale solvers for linear model predictive control. In *Proceedings of the European Control Conference (ECC)*, 128–133.

Frison, G. (2015). *Algorithms and Methods for High-Performance Model Predictive Control*. Ph.D. thesis, Technical University of Denmark (DTU).

Frison, G., Kouzoupis, D., Zanelli, A., and Diehl, M. (2017). BLASFEO: Basic linear algebra subroutines for embedded optimization. *arXiv preprint*.

Golub, G. and Loan, C. (1996). *Matrix Computations*. Johns Hopkins University Press, Baltimore, 3rd edition.

Goto, K. and Geijn, R.A.v.d. (2008). Anatomy of high-performance matrix multiplication. *ACM Trans. Math. Softw.*, 34(3), 12:1–12:25.

Griewank, A. and Walther, A. (2008). *Evaluating Derivatives*. SIAM, 2 edition.

Hairer, E. and Wanner, G. (1991). *Solving Ordinary Differential Equations II – Stiff and Differential-Algebraic Problems*. Springer, Berlin Heidelberg, 2nd edition.

Houska, B., Ferreau, H.J., and Diehl, M. (2011). An auto-generated real-time iteration algorithm for nonlinear MPC in the microsecond range. *Automatica*, 47(10), 2279–2285.

Kirches, C., Wirsching, L., Sager, S., and Bock, H. (2010). Efficient numerics for nonlinear model predictive control. In *Recent Advances in Optimization and its Applications in Engineering*, 339–357. Springer.

Mattingley, J., Wang, Y., and Boyd, S. (2010). Code generation for receding horizon control. In *Proceedings of the IEEE International Symposium on Computer-Aided Control System Design*, 985–992. Yokohama, Japan.

Mayne, D. and Rawlings, J. (2013). *Model Predictive Control*. Nob Hill.

Nocedal, J. and Wright, S.J. (2006). *Numerical Optimization*. Springer Series in Operations Research and Financial Engineering. Springer, 2 edition.

Ohtsuka, T. (2004). A continuation/GMRES method for fast computation of nonlinear receding horizon control. *Automatica*, 40(4), 563–574.

OpenBLAS (2011). OpenBLAS: An optimized BLAS library. http://www.openblas.net/.

Quirynen, R., Gros, S., and Diehl, M. (2015). Inexact Newton based lifted implicit integrators for fast nonlinear MPC. In *Proceedings of the IFAC Conference on Nonlinear Model Predictive Control (NMPC)*, 32–38.

Quirynen, R., Gros, S., Houska, B., and Diehl, M. (2016). Lifted collocation integrators for direct optimal control in ACADO toolkit. *Mathematical Programming Computation (accepted, preprint available at Optimization Online, 2016-05-5468)*.

Quirynen, R., Houska, B., and Diehl, M. (2017). Efficient symmetric Hessian propagation for direct optimal control. *Journal of Process Control*, 50, 19–28.

Van Zee, F.G. and van de Geijn, R.A. (2015). BLIS: A framework for rapidly instantiating BLAS functionality. *ACM Trans. Math. Softw.*, 41(3), 14:1–14:33.

Wirsching, L., Bock, H.G., and Diehl, M. (2006). Fast NMPC of a chain of masses connected by springs. In *Proceedings of the IEEE International Conference on Control Applications, Munich*, 591–596.