High-Performance Small-Scale Solvers for Linear Model Predictive Control

Gianluca Frison, Hans Henrik Brandenborg Sørensen, Bernd Dammann, John Bagterp Jørgensen

Abstract—In Model Predictive Control (MPC), an optimization problem needs to be solved at each sampling time, and this has traditionally limited use of MPC to systems with slow dynamic. In recent years, there has been an increasing interest in the area of fast small-scale solvers for linear MPC, with the two main research areas of explicit MPC and tailored on-line MPC. State-of-the-art solvers in this second class can outperform optimized linear-algebra libraries (BLAS) only for very small problems, and do not explicitly exploit the hardware capabilities, relying on compilers for that. This approach can attain only a small fraction of the peak performance on modern processors. In our paper, we combine high-performance computing techniques with tailored solvers for MPC, and use the specific instruction sets of the target architectures. The resulting software (called HPMPC) can solve linear MPC problems 2 to 8 times faster than the current state-of-the-art solver for this class of problems, and the high-performance is maintained for MPC problems with up to a few hundred states.

I. INTRODUCTION

In recent years, there has been an increasing interest in fast small-scale solvers for linear Model Predictive Control (MPC). This is due to both the need of extend the use of MPC to faster systems (with KHz sampling frequencies), and to the use of decomposition algorithms (where a large number of small problems has to be solved). The two main research areas in fast MPC are explicit MPC [1] and tailored solvers for on-line MPC [10]. In turn, solvers for on-line MPC can be divided into two classes: first order methods (e.g. gradient methods) and second order methods (e.g. interior-point methods). In our paper, we will focus on interior-point methods for on-line MPC, that have the useful property of converging in a number of iterations almost independent of the problems size and conditioning.

Second order methods make use of matrix-matrix linear-algebra operations (level 3 BLAS), that require $O(n^3)$ floating-point operations (flops) while using $O(n^2)$ storage space: thus each matrix element is accessed $O(n)$ times. In modern architectures the cost of a memory operation (memop) is much higher than the cost of a floating-point operation (flop). Furthermore, most instructions are pipelined, and their latency can be effectively hidden if enough independent operations are present in the code. As a consequence, an implementation of a linear-algebra routine only concerned in reducing the number of flops would attain a low performance, since the processor would be idle most of the time, waiting for operands to be fetched from main memory or for dependent instructions to complete. A technique used to mitigate both issues is blocking for registers.

High-performance implementations of level 3 BLAS can attain performances very close to the theoretical peak for large-enough matrices [4]. This performance is obtained by employing different levels of blocking (e.g. for registers, level 2 cache, etc.), copying data in contiguous memory, and using assembly code for the innermost loops and architecture-specific SIMD (Single-Instruction Multiple-Data) instructions (e.g. SSE, AVX in Intel and AMD processors). However, compilers are not very good at producing blocked code, nor at using SIMD, so this is still something that should be done by the programmer.

The drawback of the approach employed in high-performance BLAS is that, for small matrices, the cost of all these memory copies and different levels of blocking would be totally dominant. Thus, in recent years there has been much research about the possibility of improving the speed of small-scale MPC solvers, studying alternatives to BLAS.

CVXGEN [5] is a well-known small-scale solver for convex optimization problems, that can solve many MPC problems. It employs a predictor-corrector Interior-Point (IP) method, and a sparse LDL factorization for the solution of the KKT system at each iteration of the IP method. The approach used to implement the linear algebra is code generation: a tailored solver is generated for the size and the form of each individual problem. The output of the code generation process is a set of C source files, where all the single operations are written down, without loops. In a following step, the compiler has the task to convert this C code into an executable. The main advantages of this approach are that there are no loops nor function calls (and then no associated overhead) and branches (and then no branch misprediction). The main disadvantages are that instruction cache is not exploited (since each instruction is executed only once), and that the code size grows with the cube of the matrices size, becoming quickly intractable.

FORCES [2] is a numerical optimization framework for convex multistage problems, that can solve a wide class of MPC problems. It employs a predictor-corrector IP method, and a tailored solver for the KKT system, based on a block Cholesky factorization of the Schur complement of the KKT matrix. This tailored solver has been previously employed in the Fast-MPC [10] solver. FORCES uses a different approach to code generation: instead of writing down all the single operations, it uses nested triple-loops, where the loops size is tailored for each individual problem and fixed at compile time. This enables the compiler to perform loop unrolling...
when it is most profitable, while keeping the size of the executable approximately constant. The main advantage of this approach is that the performance scales much better with the problem size. The main disadvantage is that a triple-loop based approach can attain only a small fraction of the peak performance of the processor.

In this paper, we propose a novel approach to implement solvers for linear MPC problems, combining the implementation techniques of high-performance optimized BLAS libraries with the small-scale speed of code-generation and solvers specially tailored for MPC problems.

The proposed algorithm for the solution of the KKT system of MPC problems is similar to the one presented in [3], with the difference that it moves the integration process one step further: the factorization and the backward recursion of the solution are fused. This allows us to reduce the number of function calls to linear-algebra routines to 3 in the factorization and 3 in the solution, for each iteration of the Riccati-like recursion.

About the implementation, we use an approach somehow similar to the one proposed in the BLIS [9] framework, with the difference that we only block for registers and add code-generation. More specifically, we write the innermost loop as a separate (and optimized) micro-kernel, and block for the size of the registers. Furthermore, we employ a partial code-generation approach: only the two outermost loops around the micro-kernel are totally unrolled. This gives a good balance between speed and code size, and in any case a library version of the code is available too. The performance for small-scale problems is up to one order of magnitude higher than the one obtained using optimized BLAS, and the cross-over point is for problems with several hundreds states and controls, large enough for most MPC applications. Furthermore, our tests in section VI show that our solver is from 2 (for the smaller problem) to 8 (for the larger) times faster than the current state-of-the-art solver for linear MPC.

The small number of function calls means that we need to write and optimize only 6 linear algebra routines. Actually, good performance can be obtained using the reference version of the code and optimizing only the matrix-matrix multiplication micro-kernel. This approach is portable, since the only code that needs to be optimized on a new architecture is this micro-kernel. The BLIS framework will make available highly optimized micro-kernels for a number of architectures [8], and at that time it will be possible to combine our solver with these micro-kernels, to obtain high performance on an even wider range of architectures.

We will publish the HPMPC code as open-source, so our optimized solver can be used out of the box on most Intel and AMD machines.

II. PROBLEMS

In this paper, we focus our attention on efficient solvers for the Linear-Quadratic (LQ) control problem, that can be considered the core problem in MPC. In fact, it is a rather general formulation that can represent a number of problems in optimal control and estimation, and in particular it arises as sub-problem in Interior-Point (IP) methods for MPC. High-performance of a solver for the LQ control problem immediately translates in high-performance for solvers for a wider class of problems.

A. LQ control problem

The LQ control problem (LQCP) is the equality constrained quadratic program

$$\min_{u_n, x_{n+1}} \phi = \sum_{n=0}^{N-1} \varphi_n(x_n, u_n) + \varphi_N(x_N)$$

subject to

$$x_{n+1} = A_n x_n + B_n u_n + b_n$$

$$x_0 = x_0$$

where \( n \in \{0, 1, \ldots, N-1\} \) and

$$\varphi_n(x_n, u_n) = \begin{bmatrix} u_n^T & x_n^T \end{bmatrix} \begin{bmatrix} R_n & S_n & s_n \\ S_n^T & Q_n & q_n \end{bmatrix} \begin{bmatrix} u_n \\ x_n \end{bmatrix} = X_n^T Q_n X_n$$

$$\varphi_N(x_N) = \begin{bmatrix} x_N^T & p^T & \pi^T \end{bmatrix} \begin{bmatrix} x_N \end{bmatrix} = X_N^T P X_N$$

All matrices in this formulation can be dense and time variant. We assume that all matrices \( Q_n \) and \( P \) are symmetric positive definite.

B. Linear MPC problem

The linear MPC problem with linear constraints is the quadratic program

$$\min_{u_n, x_{n+1}} \phi = \sum_{n=0}^{N-1} \varphi_n(x_n, u_n) + \varphi_N(x_N)$$

subject to

$$x_{n+1} = A_n x_n + B_n u_n + b_n$$

$$x_0 = \bar{x}_0$$

$$C_n x_n + D_n u_n \geq d_n$$

$$C_N x_N \geq d_N$$

III. SOLVER FOR THE LQ CONTROL PROBLEM

In this section we want to show a solution procedure for the LQ control problem (1) equivalent to the classical Riccati recursion, but with important advantages on the implementation side.

A. Derivation

It is well known from literature that the LQ control problem can be solved by using dynamic programming. Here we do not want to repeat the proof, but only to show a procedure to optimize the stage cost that leads to an efficient implementation in practice.

The optimal stage cost at the generic stage \( n+1 \) is

$$V_{n+1}(x_{n+1}) = \begin{bmatrix} x_{n+1}^T & 1 \end{bmatrix} \begin{bmatrix} P_{n+1} & p_{n+1} \\ p_{n+1}^T & \pi_{n+1} \end{bmatrix} \begin{bmatrix} x_{n+1} \\ 1 \end{bmatrix}.$$ 

Inserting the expression

$$x_{n+1} = \begin{bmatrix} A_n & b_n \end{bmatrix} \begin{bmatrix} u_n \\ x_n \end{bmatrix}$$

and AMD machines.

even wider range of architectures.

[8], and at that time it will be possible to combine our solver with these micro-kernels, to obtain high performance on an even wider range of architectures.

We will publish the HPMPC code as open-source, so our optimized solver can be used out of the box on most Intel and AMD machines.
the optimal stage cost becomes
\[ V^*_{n+1}(x_n, u_n) = X'_n A'_n P_{n+1} A_n X_n = \]
\[ \begin{bmatrix} u_n \\ x_n \\ 1 \end{bmatrix}' \begin{bmatrix} B'_{n+1} & P_{n+1} \\ A'_{n+1} & \pi_{n+1} \end{bmatrix} \begin{bmatrix} B_n & A_n \\ 0 & 1 \end{bmatrix} \begin{bmatrix} u_n \\ x_n \\ 1 \end{bmatrix} \]
If the matrix \( P_{n+1} \) is positive definite, it can be factorized using Cholesky factorization, as
\[ P_{n+1} = L_{n+1} L'_{n+1} = \]
\[ \begin{bmatrix} L'_{n+1,12} \\ L'_{n+1,32} L'_{n+1,33} \end{bmatrix} \begin{bmatrix} L_{n+1,12} \\ L_{n+1,32} L_{n+1,33} \end{bmatrix} \]
and then the optimal stage cost becomes
\[ V^*_{n+1}(x_n, u_n) = X'_n A'_n L_{n+1} L'_{n+1} A_n X_n = \]
\[ X'_n (L'_{n+1} A_n)' (L'_{n+1} A_n) X_n \]
that can be build efficiently by exploiting the symmetry and the fact that \( L_{n+1} \) is a lower triangular matrix.

The stage cost at the stage \( n \) (dropping the indexes \( n \) and \( n+1 \) in the last equation)
\[ V_n(x_n, u_n) = \varphi_n(x_n, u_n) + V^*_{n+1}(x_n, u_n) = \]
\[ = X'_n (Q_n + (L'_{n+1} A_n)' (L'_{n+1} A_n)) X_n = \]
\[ \begin{bmatrix} u \\ x \\ 1 \end{bmatrix}' \begin{bmatrix} R + B'_{n} P_{n+1} B \& S + B'_{n} P_{n+1} A \& s + B'_{n} (P_{n+1} + p) \\ S + A'_{n} P_{n+1} A \& Q + A'_{n} P_{n+1} A \& q + A'_{n} (P_{n+1} + p) \\ 1 \end{bmatrix} \begin{bmatrix} u \\ x \\ 1 \end{bmatrix} \]
is a function of \( x_n \) and \( u_n \), and can be easily minimized with respect to \( u_n \) in the following way. The matrix is positive definite (since it is the sum of a positive definite matrix and a positive semi-definite matrix), and then the stage cost can be factorized by using the Cholesky factorization of the matrix,
\[ M_n = Q_n + (L'_{n+1} A_n)' (L'_{n+1} A_n) \]

obtaining the expression for the stage cost \( V_n(x_n, u_n) \)
\[ \begin{bmatrix} L'_{n+1,11} u_n + L'_{n+1,21} u_n + L'_{n+1,31} u_n + L'_{n+1,33} u_n \\ L'_{n+2,22} x_n + L'_{n+2,32} x_n + L'_{n+2,33} x_n + L'_{n+2,33} x_n \end{bmatrix}' \begin{bmatrix} L'_{n+1,11} x_n + L'_{n+1,21} x_n + L'_{n+1,31} x_n + L'_{n+1,33} x_n \\ L'_{n+2,22} x_n + L'_{n+2,32} x_n + L'_{n+2,33} x_n + L'_{n+2,33} x_n \end{bmatrix} \]
\[ = (L'_{n,11} u_n + L'_{n,21} u_n + L'_{n,31} u_n + L'_{n,33} u_n) (L'_{n,11} x_n + L'_{n,21} x_n + L'_{n,31} x_n) \]
\[ + (L'_{n,22} x_n + L'_{n,32} x_n + L'_{n,32} x_n + L'_{n,33} x_n) + L_n x_n + L_n x_n \]
Notice that \( u_n \) is present only in the first term of the sum: this term is a square, and then its minimum is zero, attained for the value of \( u_n \)
\[ u_n = -(L'_{n,11})^{-1} (L'_{n,21} x_n + L'_{n,31}). \]

The corresponding optimal value \( V^*_n(x_n) \) of the stage cost is
given by the remaining two terms of the sum:
\[ V^*_n(x_n) = (L'_{n,22} x_n + L'_{n,32}) (L'_{n,22} x_n + L'_{n,32}) + L_n x_n + L_n x_n \]

as in the classical formulation of the dynamic programming for the LQ control problem. Notice that the procedure gives a factorization of the matrix \( P_n \) that can be used at the following stage to efficiently compute \( A_{n-1} P_n A_{n-1} \).

The value of \( u_n \) in (4) can be rewritten as
\[ u_n = -(R_n + B'_n P_{n+1} B_n)^{-1} ((S_n + B'_n P_{n+1} A_n)x_n^+ + s_n + B'_n (P_{n+1} B_n + P_{n+1} + p_n)) = K_n x_n + k_n \]
that is the usual expression of \( u_n \) as time varying affine state feedback given by the Ricatti recursion. However, the procedure to compute \( u_n \) as in (4) is more efficient from a computational point of view. Also notice that the recursion matrix \( P_n \) of the Ricatti recursion is never computed explicitly in the above solution procedure.

**B. Algorithm**

The Ricatti-like procedure presented in the previous section leads to an efficient algorithm in practice, summarized in Algorithm 1.

The classical Ricatti recursion for the solution of the LQ control problem consist of a backward recursion for the KKT matrix factorization (with cubic complexity in the matrices size) and backward and forward substitution for the KKT system solution (with quadratic complexity in the matrices size). For small systems, the cost for the factorization and the cost for the solution have the same order of magnitude.

In the proposed algorithm the factorization and the backward substitution are fused in a single backward loop. The number of function calls to BLAS per backward iteration is only 3, thanks to the packing of matrices. This reduces the function calls overhead and the data movement.

The proposed algorithm does not contain calls to level 2 BLAS functions in the backward loop, that have been replaced by packing the vectors \( s \) and \( q \) with the matrices \( R_n, S_n \) and \( Q_n \), and the vector \( b \) with the matrices \( A_n \) and \( B_n \) to perform calls to level 3 BLAS on a single larger matrix. As a result, the calls to level 2 BLAS in the backward substitution of the the classical Ricatti recursion come almost for free, since the matrix operands are already loaded in the registers.

**Algorithm 1 Solution procedure for the LQ control problem**

1. \[ [L_{n+1,12}, L_{n+1,32}, L_{n+1,33}] \leftarrow P_1/2 \] \( \triangleright \) dpotrf
2. for \( n \leftarrow N \rightarrow 0 \) do
3. \[ L'_{n+1} \leftarrow L'_{n+1,12} B_n A_n b_n + [0 0 L'_{n+1,32}] \] \( \triangleright \) dtrmm
4. \[ M_n \leftarrow Q_n + (L'_{n+1} A_n)' (L'_{n+1} A_n) \] \( \triangleright \) dsyrk & dsyrk
5. \[ [L_{n,11}, L_{n,31}, L_{n,33}] \leftarrow M_n^{1/2} \] \( \triangleright \) dpotrf
6. end for
7. for \( n \leftarrow 0 \rightarrow N \) do
8. \[ u_n \leftarrow - 1 (L'_{n,11})^{-1} (L'_{n,21} x_n + L'_{n,31}) \] \( \triangleright \) dgemv & dtrsv
9. \[ x_{n+1} \leftarrow [B_n A_n] [u_n x_n']' + b_n \] \( \triangleright \) dgemv
10. end for

The cost of the algorithm in flops is:
\[ N((4n^3 + 4n^2 n_u + 2nx^2 + \frac{1}{2} n^3) + (\frac{13}{2} n^2 + 9nx n_u + \frac{5}{2} n^2)) \]
IV. IMPLEMENTATION DETAILS

In this section we present the techniques used in the implementation of our software.

A. Blocking for registers

The most important technique is certainly blocking for registers: this reduces the number of memops, and helps hiding the latency of operations. We will explain the idea with an example. Suppose that we want to compute the product of two squared matrices $A$ and $B$ of size $n$, and use the result to update the square matrix $C$ of size $n$:

$$ C = C + A \cdot B $$

If we use the definition of matrix-matrix product, we can compute each element $c_{ij}$ of $C$ as

$$ c_{ij} = c_{ij} + \sum_{k=0}^{n-1} a_{ik} \cdot b_{kj}. \quad (5) $$

If we store the element $c_{ij}$ in a register, the computation of one element of $C$ requires $2n$ flops ($n$ multiplications and $n$ sums) and $2n + 2$ memops (1 load and 1 store of $c_{ij}$, $n$ loads of both $a_{ik}$ and $b_{kj}$). In total, the matrix-matrix product would require approximately $2n^3$ flops and $2n^2$ memops, with a ratio flops/memops of 1.

If we use more registers to store elements of $C$, we can improve this ratio. If for example we store a $2 \times 2$ sub-matrix of $C$, then, for each $k$, we can load 2 elements of $A$ and 2 elements of $B$, to update 4 elements of $C$, as

$$ \begin{bmatrix} a_0 & a_1 \\ c_{00} + a_0 \cdot b_0 & c_{01} + a_0 \cdot b_1 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \end{bmatrix} $$

$$ \begin{bmatrix} c_{10} + a_1 \cdot b_0 \\ c_{11} + a_1 \cdot b_1 \end{bmatrix} $$

Once loaded in the registers, each element of $A$ and $B$ is used twice; the ratio flops/memops is then about 2.

In general, if we can store a sub-matrix of $C$ of size $n_r$, the ratio flops/memops is about $n_r$. In practice, the number of available registers is limited, and the size of the sub-matrix of $C$ stored in the registers has to be chosen accordingly.

The same idea can be applied to other memory levels, for example blocking for level 2 or 3 cache. However, since our target are small-scale problems that can already fit in cache, we did not implement blocking for cache, but we store the elements of the matrices in the same order as they are accessed by the matrix-matrix multiplication micro-kernel.

Furthermore, notice that the 4 multiply-accumulate in (6) are totally independent, and could be performed in parallel, while this is not the case unrolling the loop in (5). Thus blocking for registers can be used to get enough independent operations to keep the execution units busy, since most floating-point instructions are pipelined, and their throughput is lower than the latency.

B. SIMD instructions

SIMD (Single-Instruction Multiple-Data) are instruction that perform the same operation in parallel on all elements of small vectors of data. In theory, an operation on a vector of size $n_v$ can improve the performance up to $n_v$ times.

In our implementation, we make use of SSE-SSE2-SSE3 instructions (that operates on 128-bit-wide vectors, storing 2 doubles) and AVX instructions (that operates on 256-bit-wide vectors, storing 4 doubles). We mainly use the intrinsics version of the instruction: this makes the programming much easier, since the compiler takes care of registers allocation and instruction scheduling.

If we want to implement (6) using SSE3 instructions, it is

$$ \begin{bmatrix} a_0 & a_1 \\ c_{00} + a_0 \cdot b_0 & c_{01} + a_0 \cdot b_1 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \end{bmatrix} $$

where the squared brackets indicates the small vectors. As a result, the number of operations is halved.

SIMD instructions often have alignment requirements to obtain high performance: for example, SSE instructions can efficiently load and store data that is 128 bits (or 16 bytes) aligned, while for the AVX instructions the alignment requirement is 256 bits (or 32 bytes). In our LQ control problem solver, we require the data to be already aligned, and we deal with this in the IP method.

C. Customized BLAS

In order to obtain the highest performance for small problems, we implemented the few BLAS routines needed by our solver using the techniques presented above.

More in detail, we implemented a simplified version of the needed BLAS routines, with one only option per routine (e.g. we only work with lower triangular matrices). The innermost loop of each BLAS routine is implemented as a separate micro-kernel, coded using blocking for registers and SIMD.

We employ a partial code-generation approach, where the innermost loop is coded as a function (kernel), and the two outermost loops are totally unrolled. This gives a good trade-off between performance (fewer branches and indexes computation) and code size. A library version of the code is also available, and usually it is faster for system with more than about 30 states.

V. IP METHOD FOR THE LINEAR MPC PROBLEM

The linear MPC problem in (3) can be solved using an IP method. In this paper, we employ a primal-dual IP method [6]. Let us consider the general quadratic program

$$ \min_y \frac{1}{2} y' H y + g' y $$

$$ s.t. \quad A y = b $$

$$ C y \geq d $$

then at each iteration $k$ of the IP method it has to be solved a linear system of equations of the form

$$ \begin{bmatrix} H + C'(T_k^{-1} \Lambda_k) C - A' \\ -A_k \\ 0 \end{bmatrix} \begin{bmatrix} y_k \\ \pi_k \\ \sigma_k \end{bmatrix} = \begin{bmatrix} g - C'(\Lambda_k e + T_k^{-1} \Lambda_k d + T_k^{-1} \sigma \mu_k e) \\ b \end{bmatrix} $$

(7)
where $t_k$ are the slack variables, $\pi_k$ and $\lambda_k$ are the Lagrangian multipliers associated with the equality and inequality constraints, $\mu_k$ is the duality measure, $\sigma$ is a centering parameter and $e$ is a vector of ones. In the case of the linear MPC problem, it can be shown [7] that (7) is the KKT system of an instance of the LQ control problem (1). This means that at each iteration of the IP method we can use our solver for the LQ control problem to solve the linear system of equations (7), that is the main computational effort at each iteration.

VI. RESULTS

In this section we present the results of two series of test: in the one we compared the relative performance of different implementations of our solver for the LQ control problem; in the second one we compared our IP method for linear MPC with the current state-of-the-art solver for linear MPC. In case of multi-core machines, only one core is used.

A. LQ control problem

To assess the performance of the different implementations of our solver for the LQ control problem, we tested a version using for the linear-algebra BLAS and LAPACK provided by OpenBLAS 0.2.6 [11]; a version using tailored triple-loop based linear algebra and code generation; and three versions implemented using the techniques presented in this paper, and coded respectively in C code, SSE3 instructions and AVX instructions. All tests have been performed on a Laptop equipped with an Intel i5 2410M processor (2.3 GHz, up to 2.9 GHz in turbo mode), running Xubuntu 13.04; the compiler is gcc 4.7.3. In figure 1 we plot the performance in Gflops obtained using the different approaches, and compared with the theoretical peak performance of the processor (that has been computed assuming that it operates at the maximum turbo frequency, as 2.9 GHz * 2 floating-point instructions per clock (one add and one mult) * 4 flops per floating-point instruction (AVX) = 23.2 Gflops).

When our solver is linked to OpenBLAS, the performance is good for large problems (close to the theoretical peak), but it is poor for small problems. The approach making use of triple-loop and code generation is faster for small systems, but can only attain a small fraction of the theoretical peak: as a consequence, it can outperform OpenBLAS only for very small problems. The version written in C code and employing blocking for registers doubles the performance with respect to the triple-loop one.

The version using micro-kernels coded with SSE3 instructions and blocking for registers doubles again the performance. For very small problems, the performance is almost 10 times the one obtained using OpenBLAS. The version using micro-kernels coded with AVX instructions and blocking for registers almost doubles the performance again. In this test, the performance keeps increasing with $n_x$, and the maximum performance is 80.6% (18.7 Gflops) of the theoretical peak performance at turbo frequency. On the test machine, the dgemm micro-kernel has a steady performance above 90% of the peak for matrices of size up to about 340:

for larger matrices, the memory footprint exceeds L3 cache, and the performance decreases. However, this matrix size is large enough for most MPC problems.

B. Linear MPC problem

In this section we compare the IP solver part of our HPMPC toolbox with FORCES (to our knowledge the state-of-the-art solver for linear MPC), running the mass-spring test in table VI in the paper [2]. The tests are performed on several x86 and x86_64 machines, all running different flavors of Linux (mainly Ubuntu-based) and using gcc as C compiler: results are in table I.

The versions using SSE3 and AVX could be compared each other on two laptops, one equipped with the CPU Intel Core i7 3520M (Ivy Bridge) at 2.9 GHz (3.6 GHz in turbo mode), the other with the CPU Intel Core i5 2410M (Sandy Bridge) at 2.3 GHz (2.9 GHz in turbo mode). On both machines, our IP solver is from about 2 (for the smaller tests) to 8 (for the larger test) times faster than FORCES. The better performance is obtained exploiting the wider AVX instruction set. The version using OpenBLAS is faster than FORCES starting from the third problem.
We tested our code also on a laptop equipped with the older Intel Core 2 Duo P8600 at 2.4 GHz. This architecture (named Penryn) features the SSE4.2 instruction set, but not AVX. This time the maximum speed-up with respect to FORCES is about 5, lower than using AVX.

Instead the embedded system equipped with of Intel Atom Z530 in [2], we used a netbook equipped with the equivalent Intel Atom N270 processor (1.6 GHz). This architecture is different compared to all the others considered in our tests. In fact, the processor is 32-bit (and then there are only 8 SSE registers, instead of 16), the cache is smaller (24 KB L1 data cache, 512 KB L2 cache, no L3 cache), and the processor performs in-order-execution (and then the order of the instructions matters): the resulting performance is thus quite low, and it is much more difficult to write fast code. The best code was obtained using scalar SSE2 instructions (no SIMD) in inline assembly. Also in this case the maximum speed-up with respect to FORCES is about 5.

We also tested our code on a machine equipped with the AMD processors Opteron 6168 (1.9 GHz, K10 architecture, SSE3 instruction set): again, the results are similar, with a speed-up of about 6 times with respect to FORCES.

The tests show as our code implementation combines and improves two approaches: the small-scale speed of tailored solvers and code-generation, with the large-scale high-performance of optimized BLAS libraries. Furthermore, our code is better than architecture-agnostic solvers as FORCES in exploiting the advanced features of recent hardware (e.g. AVX instructions).

**VII. CONCLUSION**

In this paper, we reviewed current state-of-the-art solvers for linear MPC, and we proposed a novel approach for this class of problems. We presented a solver for the LQ control problem with good asymptotic complexity, implemented using a very small number of function calls to linear-algebra routines. This allows us to write and optimize only a small subset of BLAS, combining code-generation with high-performance techniques and exploiting the hardware specific instructions: the innermost loop of each linear-algebra routine is implemented as a separate micro-kernel, and coded using the available SIMDs. As a result, our solver outperforms both state-of-the-art linear MPC solvers and optimized BLAS libraries, attaining a performance close to the theoretical peak for a wide range of problem sizes.

**ACKNOWLEDGEMENT**

We would like to thank Gabriele Frison for the help in testing the solvers on the Atom architecture. We would also like to thank the reviewers for the useful feedback.

**REFERENCES**


**TABLE I: Run times [in ms] for 10 interior point iterations, averaged over 100 random initial states. The tests are the same as in TABLE VI in [2]. * AVX instruct. not supported.**

<table>
<thead>
<tr>
<th>M</th>
<th>n x</th>
<th>n u</th>
<th>N</th>
<th>OpenBLAS</th>
<th>SSE</th>
<th>AVX</th>
<th>FORCES</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>4</td>
<td>1</td>
<td>10</td>
<td>0.25</td>
<td>0.04</td>
<td>0.05</td>
<td>0.08</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>3</td>
<td>10</td>
<td>0.44</td>
<td>0.10</td>
<td>0.09</td>
<td>0.29</td>
</tr>
<tr>
<td>6</td>
<td>12</td>
<td>5</td>
<td>30</td>
<td>1.59</td>
<td>0.71</td>
<td>0.53</td>
<td>1.67</td>
</tr>
<tr>
<td>11</td>
<td>22</td>
<td>10</td>
<td>10</td>
<td>1.63</td>
<td>0.88</td>
<td>0.61</td>
<td>2.90</td>
</tr>
<tr>
<td>15</td>
<td>30</td>
<td>14</td>
<td>10</td>
<td>2.63</td>
<td>1.82</td>
<td>1.18</td>
<td>6.70</td>
</tr>
<tr>
<td>30</td>
<td>60</td>
<td>29</td>
<td>30</td>
<td>28.02</td>
<td>31.20</td>
<td>19.01</td>
<td>153.02</td>
</tr>
</tbody>
</table>

| Intel Core i7 3520M (Ivy Bridge) | 2 | 4 | 1 | 10 | 0.27 | 0.07 * | 0.21 |
| Intel Core i5 2410M (Sandby Bridge) | 4 | 8 | 3 | 10 | 0.51 | 0.20 * | 0.70 |
| Intel Atom N270 (Bonell) | 6 | 12 | 5 | 30 | 2.58 | 1.27 * | 4.17 |
| AMD Opteron K10 | 11 | 22 | 10 | 10 | 2.61 | 1.42 * | 6.35 |
| Intel Atom N270 (Bonell) | 15 | 30 | 14 | 10 | 4.96 | 2.85 * | 13.88 |
| 30 | 60 | 29 | 30 | 72.02 | 49.12 | 264.12 |


