2014 IEEE International Conference on Control Applications (CCA)
Part of 2014 IEEE Multi-conference on Systems and Control
October 8-10, 2014. Antibes, France

Efficient Implementation of Solvers for Linear Model Predictive Control on Embedded Devices*

Gianluca Frison[1], D. Kwame Minde Kufoalor[2], Lars Imsland[2], and John Bagterp Jørgensen[1]

Abstract— **This paper proposes a novel approach for the efficient implementation of solvers for linear MPC on embedded devices. The main focus is to explain in detail the approach used to optimize the linear algebra for selected low-power embedded devices, and to show how the high-performance implementation of a single routine (the matrix-matrix multiplication `gemm`) can speed-up an interior-point method for linear MPC. The results show that the high-performance MPC obtained using the proposed approach is several times faster than the current state-of-the-art IP method for linear MPC on embedded devices.**

I. Introduction

Embedded Model Predictive Control (MPC) is about implementing MPC algorithms on embedded hardware. This is in contrast to the traditional approach where MPC is regarded as a high-level controller implemented in a PC or server-based technology. Due to the high computational demands of MPC and the comparably limited computational resources on embedded devices, obtaining a high performance MPC is not a trivial task. The key to overcome the challenges of embedded MPC on resource-limited devices is to employ efficient algorithms that exploit the computational performance capabilities of the target platform.

In the MPC literature, there are three approaches to obtain a fast online solution of linear MPC problems: explicit MPC, first-order methods, and second-order methods. Explicit MPC [1] exploits the fact that the solution of the MPC problem is piecewise affine over a polyhedral partition, and that it can be computed off-line for all possible initial states. As a drawback, the number of regions grows exponentially with the problem size, making this approach feasible only for problems with very few states and a short horizon.

First-order methods are variants of the gradient method [9]. The main advantages of this class of solvers are that each iteration is extremely cheap (the most expensive part is a matrix-vector multiplication), and they are easy to warm-start. First-order methods can also easily exploit sparsity of the QP problem arising from MPC and their computations are easy to parallelize. However, the number of iterations can vary orders of magnitude for different initial states. Also, the matrix-vector multiplication is memory-bounded (i.e. limited by the memory operations involved and thus the memory access speed) in modern computer architecture, and hence it can attain only a low fraction of peak performance. High-performance gradient methods have been implemented on FPGAs [7], and an efficient primal-dual first-order method for MPC is implemented on a PLC in [8].

Second-order methods (such as the interior point (IP) method proposed in this paper and active-set methods) are based on the Newton method. Making use of second-order information in the computation of the search direction, they usually need less iterations to converge, and the number of iterations does not change much with the initial state (especially for IP methods). On the other hand, each iteration requires a considerable amount of work compared to first-order methods: if BLAS is used, level 3 BLAS is required. This means that the linear algebra is more complex, but it can be optimized to attain a large fraction of peak performance on modern processors, and to take advantage of multiple cores. Implementation itself is thus very important for this class of solvers.

In this paper we explicitly target embedded devices. Thanks to the widespread diffusion of mobile computing, there is a race to build increasingly faster, low-power processors. In particular, the mass-market of smartphones has the potential to provide the scientific community with plenty of cheap and powerful embedded devices. In this paper, we consider three architectures: the low-power x86 Intel Atom (found in many netbooks), the ARM Cortex A9 (found in many smartphones and development boards), and the PowerPC 603e (an old architecture, but still present in many embedded devices for control). This paper has two key contributions: it explains how to optimize the linear algebra for these embedded devices; and it shows that a high-performance implementation of a single routine (the matrix-matrix multiplication `gemm`) can speed-up the entire IP method for the linear MPC. The resulting software (that we call HPMPC) is several times faster than the current state-of-the-art IP method for linear MPC on embedded devices, FORCES [2], for a suite of benchmark test problems.

The paper is organized as follows. Section II introduces the linear MPC problem and the unconstrained sub-problem. Section III summarizes the main implementation techniques employed to optimize the `gemm` micro-kernel. Section IV describes in detail the test architectures. Section V contains the results of comparison tests with FORCES, and finally section VI contains the conclusion.

[1] Technical University of Denmark, DTU Compute - Department of Applied Mathematics and Computer Science, DK-2800 Kgs Lyngby, Denmark. {giaf,jbjo}@dtu.dk

[2] Department of Engineering Cybernetics, Norwegian University of Science and Technology, O.S. Bragstads plass 2D N-7491 Trondheim, Norway. {kwame.kufoalor, lars.imsland}@itk.ntnu.no

* The research leading to these results has benefited from collaboration within the European Union's Seventh Framework Programme under ECGA No. 607957 TEMPO – Training in Embedded Predictive Control and Optimization. The financial support offered by the Research Council of Norway and Statoil for D. K. M. Kufoalor's research work is also gratefully acknowledged.

II. Problems

A. LQ control problem

The LQ control problem (LQCP) is formulated as

$$\begin{aligned} \min_{u_n, x_n} \quad & \phi = \sum_{n=0}^{N-1} \varphi_n(x_n, u_n) + \varphi_N(x_N) \\ s.t. \quad & x_{n+1} = A_n x_n + B_n u_n + b_n \\ & x_0 = \bar{x}_0 \end{aligned} \tag{1}$$

where $n \in \{0, 1, \ldots, N-1\}$ and $\varphi_n(x_n, u_n)$ and

$$\begin{aligned} \varphi_n(x_n, u_n) &= \begin{bmatrix} u_n \\ x_n \\ 1 \end{bmatrix}' \begin{bmatrix} R_n & S_n & s_n \\ S_n' & Q_n & q_n \\ s_n' & q_n' & \rho_n \end{bmatrix} \begin{bmatrix} u_n \\ x_n \\ 1 \end{bmatrix} = \mathcal{X}_n' \mathcal{Q}_n \mathcal{X}_n \\ \varphi_N(x_N) &= \begin{bmatrix} u_N \\ x_N \\ 1 \end{bmatrix}' \begin{bmatrix} 0 & 0 & 0 \\ 0 & P & p \\ 0 & p' & \pi \end{bmatrix} \begin{bmatrix} u_N \\ x_N \\ 1 \end{bmatrix} = \mathcal{X}_N' \mathcal{P} \mathcal{X}_N \end{aligned} \tag{2}$$

All matrices can be dense and time variant. We assume that the matrices $\mathcal{Q}_n$ and $\mathcal{P}$ are symmetric positive definite.

B. Linear MPC problem

Using the same definitions in (1) and (2), the linear MPC problem with linear constraints is the quadratic program

$$\begin{aligned} \min_{u_n, x_n} \quad & \phi = \sum_{n=0}^{N-1} \varphi_n(x_n, u_n) + \varphi_N(x_N) \\ s.t. \quad & x_{n+1} = A_n x_n + B_n u_n + b_n \\ & x_0 = \bar{x}_0 \\ & C_n x_n + D_n u_n \geq d_n \\ & C_N x_N \geq d_N \end{aligned} \tag{3}$$

III. Implementation techniques

This section briefly describes the main optimization techniques used to implement our code: more details can be found in [3].

A. Computation bottleneck: `gemm` micro-kernel

In this paper, we employ an interior-point (IP) method for the solution of the linear MPC problem (3). The main computational bottleneck in an IP method is typically the computation of the search direction. In the linear MPC problem, this can be computed by solving LQCPs in the form (1). In turn, the LQCPs can be solved by means of the Riccati-like recursion presented in [4], where the most expensive linear algebra routines are part of level 3 BLAS.

All cubic operations in level 3 BLAS can be implemented as two loops around a `gemm` (general matrix-matrix multiplication) micro-kernel, following the approach proposed in [10]. The `gemm` computes $\mathcal{C} := \mathcal{A}\mathcal{B} + \mathcal{C}$, where $\mathcal{C} \in \mathbb{R}^{m \times n}$, $\mathcal{A} \in \mathbb{R}^{m \times k}$, and $\mathcal{B} \in \mathbb{R}^{k \times n}$. This micro-kernel corresponds to the innermost loop in the classical triple-loop implementation of level 3 BLAS, and it is the only part of the code that needs to be carefully optimized for the specific architecture. In our implementation, all the rest of the linear MPC solver is built around this assembly-coded micro-kernel, that is used for the most expensive computations.

B. Blocking for registers

This is the most important optimization and has a dual aim: reduce memory movements and hide operations latency.

In modern architectures, the cost (time) to move data (in the following memop, memory operation) from main memory to the CPU is much higher than the cost to perform a floating-point operation (flop). Level 3 BLAS performs $\mathcal{O}(n^3)$ flops on $\mathcal{O}(n^2)$ data, and thus every matrix element is accessed $\mathcal{O}(n)$ times: if each access needs a fetch from main memory, the implementation is memory-bounded. On the other hand, if faster memories such as registers and cache are exploited to reuse data, the flops/memops ratio increases.

In our implementation we block for registers, and an $m \times m$ sub-matrix stored in the registers increases the flops/memops ratio by a factor m. We do not block for cache, since this requires further information like Translation Look-aside Buffer (TLB) entry capacities [6], and improves the performance only for large matrix sizes, while embedded MPC problems are usually small/medium in size.

About hiding operations latency, in modern architectures floating-point operations are pipelined, and thus typically an instruction can be issued at every clock cycle (throughput), but the result is available only after a certain number of clocks (latency). As a result, instructions can be performed 'in parallel' keeping the pipeline busy only if there are no dependencies between them. Blocking for registers can be used to have enough independent instructions to hide latency and keep the floating-point units busy.

C. Use of contiguous memory

The use of contiguous memory helps exploiting the available memory bandwidth and improves cache reuse. When an element is fetched from memory, data is moved into cache in chunks (called cache lines) of typically 32 or 64 bytes. This means that the access to immediately following elements is faster, since the corresponding cache line is already in cache, and there is no need to fetch a new cache line for each element. A technique used to better exploit cache is packing of matrices such that elements are stored in memory in the same order as they are accessed by the `gemm` micro-kernel.

D. Use of SIMD

Many modern architectures have single-instruction multiple-data (SIMD) instruction sets: e.g. Intel and AMD have SSE and AVX instruction sets, while ARM has NEON. These are instructions that operate on small vectors of m elements, improving the performance up to m times (if the code is not memory-bounded). Regarding the machines considered in this paper, Intel Atom has the 4-wide SSE, and ARM Cortex A9 the 4-wide NEON, both capable of operating on small vectors of 4 floats, while doubles are processed as scalars. Thus the theoretical peak performance is higher in single than in double precision [5]. The AltiVec SIMD instruction set is available for some PowerPCs, but not for the PowerPC 603e. Compilers are often not very good in automatic vectorization, so there is an advantage in making explicit use of SIMD.

E. Target: embedded devices

The target processors are embedded devices. These are typically low cost and low power machines, that lack advanced features. This means that lower-level details of the architecture should be considered in the implementation. Tested processors lack out-of-order execution and register renaming, so instruction scheduling matters and should be carefully chosen by writing the micro-kernel code in inline assembly (or hope that the compiler makes a good job). They lack hardware prefetch as well, so software prefetch should be used to help hiding latency of L2 cache or main memory.

IV. Test architectures

A. Intel Atom

In this paper we consider the original Intel Atom processor (Bonnell micro-architecture). This is a relatively recent x86 architecture (2008), but has many features typical of older architectures, used to reduce the power consumption.

Our test machine is a netbook equipped with the popular N270 processor. It is a 32-bit processor with a thermal design power (TDP) of 2.5 W, and runs at 1.6 GHz; there are 24 KB L1 data cache, 32 KB L1 instruction cache and 512 KB L2 cache. The processor supports hyper-threading and has the SSE, SSE2 and SSE3 instruction sets. It is an in-order processor (i.e. instructions are performed in the same order as in the source code).

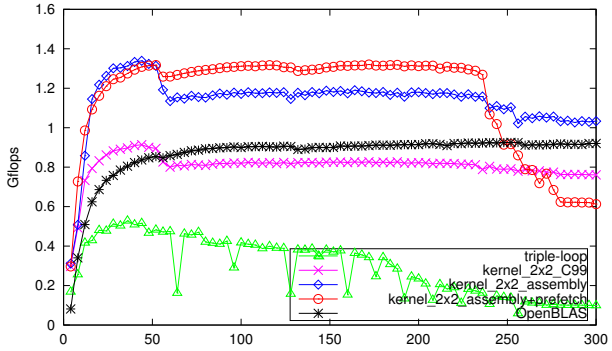
A 32-bit x86 processor has 8 floating-point registers: 4 of them are used to store a sub-matrix of $\mathcal{C}$, while the other 4 are to store elements from $\mathcal{A}$ and $\mathcal{B}$ and intermediate results.

The double precision SIMD are implemented in the SSE2 instruction set that can operate on small vectors of 2 doubles. However, in the Bonnell architecture the SSE2 vector multiply instruction is implemented as two sequential scalar multiplies. This means that SSE2 SIMD is actually slower than scalar code, since scalar instructions can be better reordered. Thus the best performance is obtained using scalar code: 4 registers are used to store a 2×2 sub-matrix of $\mathcal{C}$, and then a 2×2 micro-kernel is chosen.

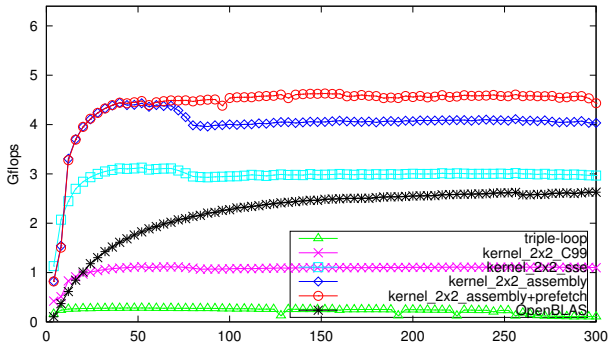
The single precision SIMD are implemented in the SSE instruction set, that is implemented properly and can operate on small vectors of 4 floats at a time. A 4×4 micro-kernel is chosen since 4 registers can hold a 4×4 sub-matrix of $\mathcal{C}$.

SSE or SSE2 instruction sets do not support fused multiply-add, adding further instruction dependences. The Atom architecture can issue an addition every cycle, but a multiplication only every other cycle. This limits the theoretical peak performance to 1 floating-point operation per cycle because in linear algebra routines there is an equal number of additions and multiplications. At 1.6 GHz it means 1.6 Gflops (where Gflops $=$ [flops/cycle] $\cdot$ [clock in GHz]) in double precision (scalar operations) and 6.4 Gflops in single precision (4-wide SIMD).

The achieved performance is limited by the very small number of registers. The processor does not support register renaming, and thus only 4 registers are actually used to store both intermediate results and elements of $\mathcal{A}$ and $\mathcal{B}$, limiting the possibility to effectively hide latency. This limitation is in part mitigated by the fact that the x86 architecture is CISC, and one of the operands of additions and multiplication can be in memory (but all instructions take two operands, and thus one of the two is overwritten with the result).



(a) Atom dgemm.



(b) Atom sgemm.

Fig. 1: Performance test of different implementations of `gemm` for squared $n \times n$ matrices, $n \in [4, 300]$, on an Intel Atom N270. Peak performance in double (single) precision is 1.6 (6.4) Gflops.

Fig. 1 reports the performance in Gflops for different single and double precision implementations of the routine for matrix-matrix multiplication. In general, triple-loop shows a poor performance (green), while blocking for registers and packing the matrices into contiguous memory improves notably the performance (magenta). In single precision, an important performance boost is achieved by the use of 4-ways SIMD (cyan). Since the processor is in-order, the maximum performance is obtained by carefully reordering instructions in inline assembly (blue). However, when memory footprint exceeds L1 cache (for size around 50 in double precision) there is a certain degradation of performance. Using software prefetch, it is possible to keep the same performance also for matrices fitting in L2 cache (red). For larger matrices, the performance drops significantly. Fortunately, this happens for matrices that are larger than the matrices in most embedded MPC applications.

Our best kernel reaches up to 83% (1.34 Gflops) of theoretical peak performance in double precision and 72% (4.63 Gflops) in single precision. It clearly outperforms optimized BLAS libraries such as OpenBLAS [11].

B. ARM Cortex A9

The ARM architecture is a quite different architecture compared to x86. It is a RISC architecture (and thus supports fewer addressing modes), but it has a rich set of instructions and many registers, making code optimization easy.

Our test machine is a development board called Wandboard Quad. It has a quad-core ARM Cortex A9 CPU running at 1 GHz. The Cortex A9 processor supports out-of-order execution and register-renaming only for general-purpose registers, while the floating-point (FP) units perform in-order execution without register-renaming. Each core has 32 KB L1 data and instruction caches, and all cores share 1 MB of L2 cache. In this paper, we address one CPU core.

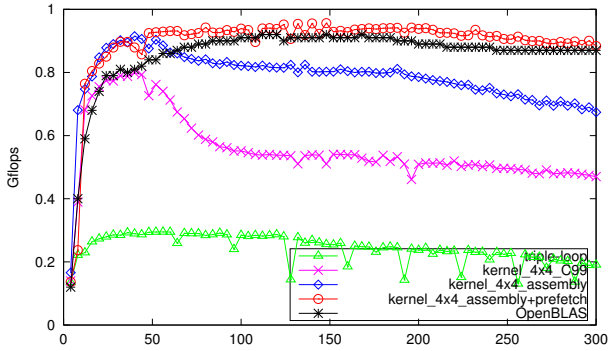
The Cortex A9 core has a scalar FP unit (VFPv3), and a SIMD unit (NEON) supporting only single-precision FP numbers (4-way SIMD). There are 32 double-word FP registers used by both VFP and NEON instructions. Each register can hold a double (registers d0-d31), while only the lower 16 registers can hold two scalar floats each (register s0-s31, where e.g. s0 and s1 are the lower and upper half of d0). Couples of consecutive d registers can be used to hold 128-bit wide vectors of 4 floats (registers q0-q15, where e.g. d0 and d1 are the lower and upper half of q0). As a result, there are 32 scalar registers (giving a 4×4 kernel for both scalar double and single precision), and 16 4-wide NEON registers (giving a 8×4 kernel for vector single precision).

The VFPv3 can operate on scalar doubles and floats. It supports fused multiply-add (FMA). In double precision, it can perform a FMA every other clock cycle, while in single precision it can perform a FMA every clock cycle. At 1 GHz, this gives for the VFP a theoretical peak performance of 1 Gflops in double and 2 Gflops in single precision.

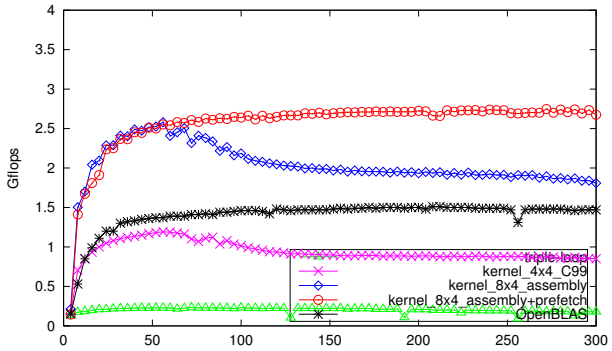
In single precision, the NEON co-processor can be used too, performing a 4-way FMA every other clock cycle, that at 1 GHz gives a theoretical peak performance of 4 Gflops.

In double precision (Fig. 2a), the large number of registers and the FMA instruction facilitate code optimization. The 4×4 kernel written in C can get about 80% if data fits in L1 cache (magenta). Reordering instructions such that memory loads are performed in idle cycles between FMAs, the performance arrives at 90%, if data fits in L1 cache (blue). The use of software prefetch gives a steady performance for data fitting in L2 cache, with maximum performance of 95%.

In single precision (Fig. 2b), if VFP is used, the performance improves with respect to double (since FMA can be performed every cycle), but does not double, since the processor seems to be unable to perform a FMA and a load in the same cycle (magenta). If NEON is used, the performance increases even more, but the best achieved performance is only about 68% of theoretical peak (blue): it looks like there is a performance penalty in mixing FMAs and loads (it is known that in the Cortex A9 there is a performance penalty mixing VFP and NEON instructions, and the two facts may be related). Again, the use of software prefetch gives a steady performance for data fitting in L2 cache (red). The achieved performance is competitive with respect to OpenBLAS (black) that does not make use of NEON.



(a) Cortex A9 dgemm.



(b) Cortex A9 sgemm.

Fig. 2: Performance test of different implementations of `gemm` for squared $n \times n$ matrices, $n \in [4, 300]$, on an ARM Cortex A9. Peak performance in double (single) precision is 1.0 (4.0) Gflops.

C. PowerPC 603e: G2_LE Core

The PowerPC target platform is the ABB AC500 PM592-ETH programmable logic controller (PLC), which has a Freescale MPC8247CVRTIEA microcontroller (SoC). The core is the G2_LE implementation of the MPC603e microprocessor. Our test PLC is equipped with 4MB RAM for user program memory and 4MB integrated user data memory.

In recent times, the increase in computational resources on PLCs and the emerging software support for the C/C++ programming language motivate the PLC implementation of optimization-based algorithms (e.g. MPC). However, the ABB PLC is a typical example of a target platform where the programmer's options for microprocessor performance exploitation are limited by a restricted list of system programming and runtime support libraries (implemented in a firmware API). The PLC software development tool, ABB PS501 Control Builder Plus 2.3 (based on the CoDeSys platform), provides programming and runtime support configurations for ANSI C89 and C99 code integrated into a PLC software/runtime architecture with a restricted set of C standard library functions. Therefore the PLC C code application consists of an IEC 61131-3 function or function block written in C. The GNU GCC 4.7.0 compiler toolchain is used to compile the C code part of the PLC application. Linking against external libraries (binaries) is not supported, implying that a library-free C code is required.

The G2_LE core is a low-power (1.5W) 32-bit RISC processor running at 400 MHz. It is equipped with independent on-chip 16 KB L1 caches for instructions and data, and on-chip memory management units (MMUs).

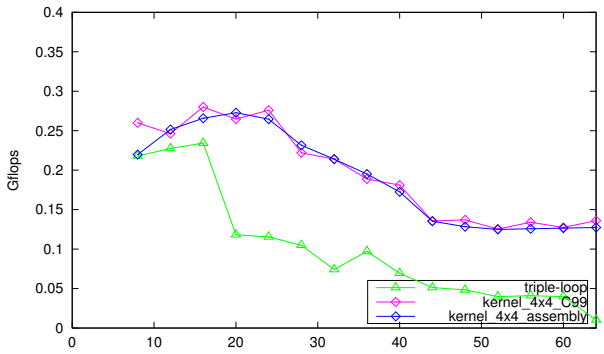
Despite the low-power design, the PowerPC G2_LE core can execute instructions out-of-order. Performance is further boosted by the superscalar architecture. A pipelined FP unit for all single-precision and most double-precision operations is also implemented, and there are 32 64-bit FP registers, each holding a single or double precision operand.

A single-precision FMA can be issued every clock cycle, whereas its double-precision counterpart every other cycle. Single-precision FMA instructions, therefore, operate faster than double-precision ones. Note that our PowerPC does not enjoy the luxury of having a floating-point SIMD instruction set, which was carefully exploited for performance boost in the Intel Atom and the ARM Cortex A9.

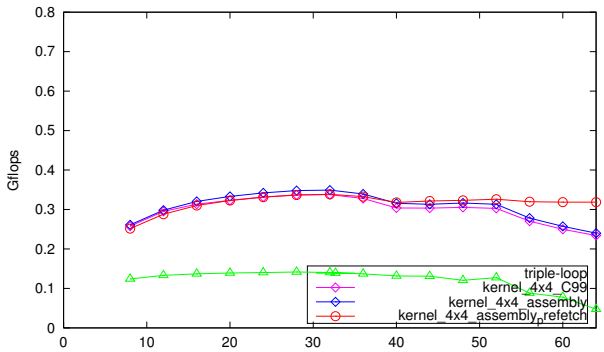
The features of the G2_LE core presented above suggest a theoretical peak of 0.4 Gflops in double and 0.8 Gflops in single-precision when the CPU is running at 400 MHz. We choose a 4×4 kernel for both double- and single-precision, and the tests results are presented in Fig. 3.

In double precision (Fig. 3a), the triple-loop version (green) can attain a good performance only for very small matrices, and performance drops significantly when the data has to be fetched from main memory (and especially for matrix size multiple of 32, due to the 4-way associativity of cache). The use of a 4×4 kernel gives a slight performance boost for matrices fitting into cache, but more importantly, it helps considerably when the memory footprint exceed cache size, since every element of $\mathcal{A}$ and $\mathcal{B}$ is used 4 times once in registers. Interestingly, the assembly coded kernel does not improve performance: FMA and the large number of registers make optimization easy, so `gcc` with -O2 already produces good code. Nevertheless, an optimized assembly code helps in case the overall code cannot be compiled with optimization flags. There are no advantages using prefetch. The maximum performance is 0.28 Gflops (70% of peak).

In single precision (Fig. 3b), our tests give a quite different picture compared to double precision. The first impression is that the performance graphs are much flatter, without the typical performance peak for data fitting in cache: the best attained performance is 0.349 Gflops (43.6% of theoretical peak). Our tests point toward the instruction fetching as the bottleneck: in fact, for $n = 32$, leaving only FMAs in the kernel loop coded in assembly, the performance ramps up to 0.45 Gflops, but leaving only memory operations the kernel execution time halves again, so memory movement is not the bottleneck either. The G2_LE core reference manual reports that the core can sustain 2 instruction fetches per clock cycle, and a memory load and a FMA can execute in parallel every clock cycle. In practice, however, the fact that the combination of loads and FMAs is slower than each of them alone is a strong argument that the core cannot co-issue load and FMA. In this framework, the performance gain of kernels compared to triple-loop is due to the lower number of memory instructions, rather than memory movements.



(a) PowerPC dgemm.



(b) PowerPC sgemm.

Fig. 3: Performance test of different implementations of `gemm` for squared $n \times n$ matrices, $n \in [8, 64]$, on an PowerPC 603e. Peak performance in double (single) precision is 0.4 (0.8) Gflops.

V. Results

In this paper we employ an interior-point (IP) method for the solution of the linear MPC problem. The current version of the software (that we call HPMPC, for High-Performance implementation of solvers for MPC) is a primal-dual IP, supporting box constraints on both inputs and states. The search direction is found by solving the LQCP (1) using the Riccati-like algorithm proposed in [4].

The key element of our implementation is that the linear-algebra routines in the LQCP solver are built around the optimized `gemm` micro-kernel. In this way, most of the computations are performed using a highly-optimized routine, tailored for the specific architecture, and the performance advantage with respect to triple-loop based implementations increases with the problems size.

In this section, we compare the performance of our software with the current (to our knowledge) state-of-the-art IP solver for linear MPC on embedded devices, FORCES [2]. FORCES makes use of code generation to build a solver tailored for the special problem instance: in particular, the linear-algebra routines are implemented as triple-loops, where the loop size is fixed, and thus the compiler can unroll the code where profitable, and perform other optimizations.

The constrained MPC problem for the comparison tests entails the control of a chain of M number of masses interconnected by springs.The problem size is defined by

TABLE I: Run times [in ms] for 10 IP iterations. The tests are the same as in TABLE VI in [2]. #: problem data too big to fit in RAM.

			Intel Atom N270 @ 1.6 GHz				ARM Cortex A9 @ 1.0 GHz				PowerPC 603e @ 0.4 GHz					
			HPMPC -O3		FORCES -O3		HPMPC -O3		FORCES -O3		HPMPC -O1		HPMPC no opt.		FORCES no opt.	
n_x	n_u	N	double	single	double	single	double	single	double	single	double	single	double	single	double	single
4	1	10	0.48	0.46	1.21	0.99	0.52	0.40	1.14	0.93	3.86	2.13	6.61	4.82	16.77	14.56
8	3	10	1.30	1.00	3.94	3.06	1.43	1.03	4.23	3.40	10.25	6.11	15.75	11.85	53.68	47.02
12	5	30	7.58	5.49	26.32	19.60	10.62	7.02	28.29	22.56	64.38	41.92	96.68	74.08	327.15	288.04
22	10	10	9.18	5.14	36.79	25.24	12.73	7.22	39.96	33.54	70.18	44.92	98.10	72.85	496.25	437.93
30	14	10	18.30	9.20	75.46	56.56	25.86	13.45	121.71	70.88	143.23	90.85	189.61	137.10	1120.54	988.38
60	29	30	332.72	130.59	1717.60	1468.38	531.34	225.07	1876.28	1373.46	#	#	#	#	#	#

$n_x = 2M$ states, $n_u = M-1$ control inputs, and the horizon N. The benchmark problem instances and the same test data used in [2] were prepared for our embedded platforms, table I summarizes the results. The same optimization flags have been used for both HPMPC and FORCES.

In the cases of Intel Atom N270 and ARM Cortex A9, the tests were easy: they run Ubuntu based operating systems (the compiler is gcc 4.6.3), and both solvers worked without any hacking. The optimization flags are `-O3 -msse3 -mfpmath=sse -march=atom` for the Atom, and `-O3 -marm -mfloat-abi=softfp -mfpu=neon -mcpu=cortex-a9` for the Cortex A9. For both architectures, the picture is alike: HPMPC is better in exploiting the processors capabilities, especially in single precision (where SIMD can be used), and the speed-up increases with the problem size, from 2x for the smaller problem up to 6x for the largest.

In case of the PowerPC 603e PLC, the tests were much harder. The first limitation is that the code has to be library-free and consist of a single C source file. Regarding HPMPC, the required hacking consisted of adding all needed files as 'headers' to the 'main'. Besides setting the compiler option `-mcpu=603e`, the maximum optimization level that could be used is `-O1`. In case of FORCES, many more hackings were needed to make the code work. The limitation that required most work in preparing the FORCES generated C code for the PLC is related to initialization of pointers. A pointer cannot be initialized by the address of another variable during declaration. That is, instructions like `float* myFloat_2 = &myFloat_1;` are not allowed. In the FORCES code, this means over 200 variable definitions must be modified, and a new function was written for initializing (setting up) all the pointers in the required format. Also, since the FORCES code generator is not open-source, the above modifications had to be done manually for each problem instance. At the end, FORCES could run on the PLC, but without using any optimization flag, not even `-O1`. Table I presents the results of 3 tests on the PLC: FORCES (without optimization), HPMPC with `-O1`, and for comparison HPMPC without optimization. The assembly coded `gemm` kernel gives HPMPC a good performance even without any optimization flag, and with `-O1` gives a speed-up from 7x to 10x compared to FORCES.

VI. Conclusion

In this paper, we proposed a novel approach to implement solvers for linear MPC on embedded devices.

We presented implementation techniques for level 3 BLAS linear algebra based on the use of optimized micro-kernels, and compared its performance to classical triple-loop based linear algebra. Furthermore, we described in detail how to optimize these micro-kernels for three different embedded architectures: Intel Atom, ARM Cortex A9 and PowerPC G2. We could get similarly high performance-per-GHz for the former two, despite being deeply different. For the latter, we could not get such a performing implementation, but we could point out the likely architectural bottleneck, and still improve the performance considerably compared to the triple-loop version.

The optimized micro-kernels have been used as the horse-power for an IP method for linear MPC. A comparison of our solver with the current state-of-the-art interior-point for MPC on embedded devices shows a considerable speed-up.

References

[1] A. Bemporad, M. Morari, V. Dua, and E.N. Pistikopoulos. The Explicit Linear Quadratic Regulator for Constrained Systems. *Automatica*, 38(1):3–20, January 2002.

[2] A. Domahidi, A. Zgraggen, M.N. Zeilinger, M. Morari, and C.N. Jones. Efficient interior point methods for multistage problems arising in receding horizon control. In *IEEE Conference on Decision and Control (CDC)*, pages 668 – 674, Maui, HI, USA, December 2012.

[3] G. Frison, H.H. B. Sørensen, B. Dammann, and J.B. Jørgensen. High-performance small-scale solvers for linear model predictive control. In *IEEE European Control Conference*, pages 128–133. IEEE, 2014.

[4] G. Frison and J.B. Jørgensen. Efficient implementation of the riccati recursion for solving linear-quadratic control problems. In *IEEE Multi-conference on Systems and Control*, pages 1117–1122. IEEE, 2013.

[5] G. Frison, L.E. Sokoler, and J.B. Jørgensen. A family of high-performance solvers for linear model predictive control. In *IFAC World Congress*, pages 3074–3079. IFAC, 2014.

[6] Kazushige Goto and Robert A. van de Geijn. Anatomy of high-performance matrix multiplication. *ACM Trans. Math. Softw.*, 34(3), 2008.

[7] J.L. Jerez, P.J. Goulart, S. Richter, G. Constantinides, E.C. Kerrigan, and M. Morari. Embedded Predictive Control on an FPGA using the Fast Gradient Method. In *European Control Conference*, pages 3614 – 3620, Zurich, Switzerland, 2013.

[8] D. K. M. Kufoalor, S. Richter, L. Imsland, T. A. Johansen, M. Morari, and G. O. Eikrem. Embedded Model Predictive Control on a PLC using a Primal-Dual First-Order Method for a Subsea Separation Process. In *22nd IEEE Mediterranean Conference on Control and Automation*, Palermo, Italy, 2014.

[9] M. Morari S. Richter and C.N. Jones. Towards Computational Complexity Certification for Constrained MPC Based on Lagrange Relaxation and the Fast Gradient Method. In *IEEE Conference on Decision and Control*, pages 5223 – 5229, 2011.

[10] Field G. Van Zee and Robert A. van de Geijn. BLIS: A framework for generating BLAS-like libraries. FLAME Working Note #66. Technical Report TR-12-30, The University of Texas at Austin, Department of Computer Science, Nov. 2012. submitted to ACM Transactions on Mathematical Software.

[11] Xianyi Zhang. Openblas. http://www.openblas.net/.